Lisp

Teil 1

Volker Bauche, Dr. Horst Friedrich, Berlin

Die Programmiersprache Lisp (Abkürzung für List Processing) wurde bereits Ende der 50er Jahre etwa zeitgleich mit Fortran entwickelt und zählt somit zu den ältesten Programmiersprachen überhaupt. Sie diente ursprünglich hauptsächlich der Symbolmanipulation, überstreicht aber heute bereits alle Anwendungsgebiete. Lisp ist im Gegensatz zu den sequentiellen und imperativen Sprachen wie Fortran, Pascal, Algol und C eine funktionale Sprache, die in ihrer Philosophie stark von anderen Sprachen abweicht und durch ihre Klarheit und ihr einheitliches Konzept besticht. Das soll nicht heißen, daß Lisp all diese Sprachen ersetzen soll, aber man sollte diese hochentwickelte Sprache bei der Auswahl der Programmiersprache besonders für anspruchsvollere Aufgaben mit in Erwägung ziehen. Viele in den anderen genannten Sprachen aufwendig umzusetzende Programmiertechnologien lassen sich in Lisp sehr viel effektiver anwenden. Dazu zählen unter anderem Methoden der Künstlichen Intelligenz, etwa der Wissensrepräsentation oder des Wissenserwerbs. Ein großer Vorteil der Sprache liegt in der später noch ausführlicher zu erläuternden formalen Nicht-Unterscheidbarkeit von Daten und Programmen in Lisp.

Internationale Untersuchungen haben ergeben, daß sich etwa 60 Prozent aller Lisp-Anwender mit Expertensystemen beschäftigen. Typische Einsatzgebiete sind die Planung (26 % der Lisp-Anwender), die Softwareentwicklung (21 %), die Interpretation von Signalen (17 %), die Robotertechnik (17 %), das Gebiet des CAD/CAE (16 %), die Verarbeitung natürlicher Sprache (14 %), die Diagnose (12 %), das Training und die Simulation (12 %) und die industrielle Automatisierung (11 %), wobei sich die Einsatzgebiete überschneiden (z. B. Robotertechnik und Planung).
In diesem Kurs werden Sie Beispiele kennenlernen, in denen besonders die Vorteile der Sprache sichtbar werden. Als Test könnten Sie versuchen, das eine oder andere vorgestellte Beispiel in einer anderen Programmiersprache Ihrer Wahl zu schreiben.

Common Lisp entstand Anfang der 80er Jahre, nachdem sich Lisp in viele Dialekte aufgespalten hatte, und wird heute als Quasi-Industriestandard akzeptiert. Weltweit arbeiten mehrere Gruppen an einer weiterführenden Lisp-Standardisierung.

Als Grundlage für diesen Kurs wurde eine Teilmenge von Common Lisp gewählt. Eine vollständige Implementation von Common Lisp wurde am Zentralinstitut für Kybernetik und Informationsprozesse der AdW vorgenommen und ist unter der Bezeichnung EXPERT Common LISP (XCL) für den K 1840 und für den P 8000 verfügbar. Diese Implementation ist für den K 1840 über Robotron-Projekt Dresden, oder über das ZKI der AdW der DDR zu beziehen. Die Version für den P 8000 ist nur über das ZKI erhältlich. Alle angeführten Beispiele können jedoch auch unter Golden Common LISP von Gold Hill Computers, mit einigen Namensänderungen unter PC-Scheme (sprich: Skiem) von Texas Instruments und ein großer Teil von ihnen unter MuLisp von Microsoft abgearbeitet werden. Diese Systeme laufen auch auf dem EC 1834, XT- und AT-Rechnern. Für diese Rechner existieren außerdem die Systeme TLC-LISP (von The Lisp Company) und IQ-LISP (von Integral Quality). Das System X-LISP ist in C geschrieben und als Public-domain-Software im Quelltext verfügbar. Für den A 7150 ist von den genannten Lisp-Systemen Golden Common Lisp (Gold Hill Computers) verfügbar. Das MuLisp für U 880-Rechner bzw. für den A 7100 unterscheidet sich dagegen sehr stark von dem hier vorgestellten Lisp-Dialekt.

In diesem Lisp-Kurs werden Übungen und Aufgaben angegeben. Zum besseren Verständnis von Lisp sollten Sie die Übungen mit einem Lisp-System am Computer abarbeiten. Die Lösungen der Aufgaben finden Sie am Ende des Teils, in dem sie gestellt wurden.

Symbolische Ausdrücke

Die symbolischen Ausdrücke oder *S-Ausdrücke* sind die grundlegenden Datenstrukturen in Lisp. Sie können einerseits als Daten und andererseits als Prozeduren interpretiert werden. Da beide formal nicht unterscheidbar sind, eröffnet sich dem Lisp-Nutzer die Möglichkeit, scheinbar komplizierte Probleme mit verblüffender Einfachheit zu kodieren. So kann ein Programm andere ohne weiteres als Daten benutzen, also auch erzeugen, verändern und aktivieren.

Lisp stellt eine Vielzahl von Datentypen bereit. Im Gegensatz zu konventionellen Programmiersprachen wird in Lisp nicht den Variablen, sondern den Daten ein Typ zugeordnet, so daß eine Lisp-Variable beliebige Daten aufnehmen kann. Das Typsystem basiert auf einer Untermengenhierarchie, in die sowohl alle vordefinierten als auch die vom Nutzer definierten Typen eingeordnet werden.

Atome

Von den atomaren Datentypen werden hier nur Symbole, ganze Zahlen, Gleitkommazahlen, Zeichenketten und in einem späteren Lehrgang Strukturen vorgestellt. Common Lisp enthält außerdem noch Zeichen, Vektoren, Felder, Hash-Tabellen, Pakete, Ströme, Pfadnamen, Gleitkommazahlen unterschiedlicher Genauigkeit, gebrochene Zahlen, komplexe Zahlen sowie Zufallszahlen.

Ganze Zahlen (Integer) sind Aneinanderreihungen von Ziffern eventuell auch mit einem führenden Vorzeichen (+ oder −), so beispielsweise
−0, 0, +6, 1024 *und*
15511210043330985984000000
(die Fakultät von 25).

Gleitkommazahlen (Float) bestehen aus einer Folge von Ziffern, die einen Dezimalpunkt enthält. Sie können mit einem Vorzeichen und mit einem Exponenten notiert werden. Beispiele für Gleitkommazahlen sind:
2.0, +6.02e23,
3.14159265358979323 84 (pi).

Zeichenketten sind beliebige Zeichenfolgen, die in zwei Anführungszeichen (") eingeschlossen werden, beispielsweise "z", "Zeichenkette" und "Common LISP".

Symbole sind Aneinanderreihungen von Buchstaben, Zahlen und Sonderzeichen (mit Ausnahme der Zeichen **(**, **)**, ‘, ’, ”, ;, *Komma* und *Leerzeichen*). Die Symbole sind mit den Bezeichnern anderer Programmiersprachen vergleichbar. Lisp-Symbole sind zum Beispiel: X, atom, +, −, NIL, K17, 10-B (Man beachte: −1 ist eine Zahl und 1− ein Symbol). Das Lisp-Symbol wandelt alle Kleinbuchstaben aus Symbolnamen in Großbuchstaben um, so daß x und X dasselbe Symbol bezeichnen.

Listen

Eine Liste wird in Lisp durch das Einschließen von Objekten in eine öffnende und eine schließende Klammer notiert. Da Listen selbst wieder Objekte von Lisp sind, gibt es somit Listen, die Listen enthalten.

(1 2 3) eine Liste mit den Elementen 1, 2 und 3
(a (1 3) b) eine Liste mit den Elementen a (ein Symbol), (1 3) (eine Liste) und b (ein Symbol)

Eine spezielle Liste ist die leere Liste. Sie kann durch eine öffnende und eine schließende Klammer, also **()**, oder durch das Symbol NIL notiert werden. Das Semikolon kennzeichnet den Beginn eines Kommentars, der durch das Zeilenende begrenzt ist.

Die erste Lisp-Sitzung

Ehe man beginnt, in einem Lisp-System zu arbeiten, sollte man sich dessen grundsätzliche Arbeitsweise klarmachen. Ein Lisp-System arbeitet interpretativ: Ein Lisp-Ausdruck (S-Ausdruck) wird eingelesen, ausgewertet (der Wert des Lisp-Ausdrucks berechnet) und anschließend das Ergebnis ausgegeben. Wird durch den Lisp-Interpreter ein Fehler entdeckt, gerät man in einen sogenannten Break-Status. Man sollte vor dem Arbeiten in der Dokumentation nachlesen, wie man in diesem Falle im jeweiligen System weiterarbeiten kann (z. B. durch Eingabe von :r in XCL, t in MuLisp, <CTRL>C in GC-LISP, <CTRL>Q in PC-Scheme).

Nachdem das Lisp-System gestartet wurde, zeigt es mit einem Promptzeichen (hier das Zeichen >) seine Eingabebereitschaft an:

Starten des Lisp-Systems

>	Ausgabe des Promptzeichens
>**3**	Eingabe des Objektes 3, dieses wird ausgewertet
3	Ausgabe des Ergebnisses der Auswertung

>() nun Eingabe einer leeren Liste
NIL Ergebnis der Auswertung
Beenden der ersten Lisp-Sitzung

Zahlen werden als Konstanten betrachtet und liefern bei Auswertung sich selbst als Wert. Konstanten sind auch die Symbole T und NIL (das auch als Zeichenfolge **()** eingebenen werden kann). NIL wird immer zu NIL und T immer zu T ausgewertet.

Übung 1
In der ersten Lisp-Sitzung sollten Sie sich durch die Eingabe von Zahlen, Zeichenketten und der Symbole NIL und T mit der Arbeitsweise des Interpreters vertraut machen. Provozieren Sie durch Eingabe der Liste (1 22 33) einen Fehler, damit Sie in den Break-Zustand des Lisp-Systems gelangen, und verlassen Sie diesen anschließend. Mit (exit) oder (system) können Sie die Lisp-Sitzung beendet werden.

Formen und Funktionen

Um richtige Berechnungen (Auswertungen) in Lisp durchführen zu können, muß man Funktionsaufrufe eingeben. Die Syntax eines Funktionsaufrufes ist sehr einfach: Ein Funktionsaufruf ist eine Liste, deren erstes Element das Symbol einer Funktion sein muß (sonst kommt es zu einem Fehler) und deren restliche Elemente passende Argumente für diese Funktion sein müssen. In Common Lisp ist + das Symbol für die Additionsfunktion:

(+ 3 6) ⇒ 9

In allen Beispielen steht die Zeichenfolge ⇒ für „wird ausgewertet zu". Der Aufruf der Funktion + mit den Argumenten 3 und 6 wird also zu 9 ausgewertet. Das Symbol + besitzt eine besondere Eigenschaft: Es ist auch als Funktion verwendbar. Die Argumente eines Funktionsaufrufes werden vor der Aktivierung der Funktion ausgewertet. In der Form (+ (* 2 8) (− 12 8)) wird zuerst (* 2 8) und dann (− 12 8) ausgewertet; die so erhaltenen Ergebnisse (16 und 4) werden der Additionsfunktion übergeben. Anschließend wird das Gesamtergebnis 20 ausgegeben.

Ein auswertbarer S-Ausdruck wird auch als Form bezeichnet. Damit man unterscheiden kann, ob eine Liste eine Form oder eine Datenliste ist, wird die Spezialform Quote verwendet. Im Gegensatz zu den Funktionen bleiben bei den Spezialformen die Argumente unausgewertet. Quote erhält ein Argument und liefert als Ergebnis das Argument ohne Auswertung zurück. Somit kann Quote verwendet werden, um die Auswertung zu verhindern.

(quote (+ 3 6)) ⇒ (+ 3 6)
(quote objekt) ⇒ OBJEKT da Lisp alle Buchstaben in Großbuchstaben umwandelt:
Da Konstanten sehr häufig verwendet werden, gibt es für (quote objekt) die abkürzende Schreibweise 'objekt.

'(+ 3 6) == (quote (+ 3 6))
⇒ (+ 3 6)
'(das ist eine liste)
== (quote (das ist eine liste))
⇒ (DAS IST EINE LISTE)

Das doppelte Gleichheitszeichen == ist dabei zu lesen als *ist identisch mit*.

Funktionen zur Listenverarbeitung

Die Grundfunktionen für die Listenverarbeitung sind Car Cdr und Cons. Car liefert das erste Element einer Liste und Cdr die Restliste ohne das erste Element. Die Funktion Cons erhält als Argument ein Objekt und eine Liste und erzeugt eine neue Liste, die aus dem Objekt als erstes Element und aus den Elementen der Liste des zweiten Argumentes besteht.

(car '(eine liste)) ⇒ EINE
(cdr '(eine liste)) ⇒ (LISTE)
(cons 'eine '(liste)) ⇒ (EINE LISTE)
(cdr '(ein-element)) ⇒ NIL

Car und Cdr liefern für die leere Liste, also für NIL, als Ergebnis wiederum NIL.

(cons 'peter nil))
== (cons 'peter ())
⇒ (PETER)
(cons nil nil) == (cons () ())
⇒ (NIL)

Da Funktionen ihre Argumente auswerten, kann man Car Cdr und Cons auch ineinandergeschachtelt verwenden:

(car (cons 'eine '(liste)))
⇒ EINE
(cdr (cons 'eine '(liste)))
⇒ (LISTE)
(car (cdr '(eine liste)))
⇒ LISTE

Man beachte, daß (car '(cdr (eine liste))) das Symbol Cdr als Ergebnis liefert, da es das erste Element der „gequoteten" Liste ist.

Sehr häufig müssen mehrere Car- und Cdr-Funktionen nacheinander ausgeführt werden, um auf ein bestimmtes Objekt einer Liste zugreifen zu können. Dazu können Funktionen der *Car-Cdr-Familie* verwendet werden. Die Namen der Funktionen dieser Familie sind Cxxr, Cxxxr und Cxxxxr, wobei jedes x entweder durch ein a (für car) oder durch ein d (für Cdr) zu ersetzen ist:

(cadr '(a b c)
== (car (cdr '(a b c)))
⇒ B
(cadadr '((a b) (c d) (e f))) ==
(car (cdr (car (cdr '((a b) (c d) (e f))))))
⇒ D

Die Reihenfolge der Anwendung von Car und Cdr ist genau umgekehrt zu der Reihenfolge der Notation von d und a in den Namen der Funktionen.

Übung 2
Durch Anwendungen von Funktionen der CAR-CDR-Familie soll auf das Symbol S in den folgenden Listen zugegriffen werden:
2a) (L I S P)
2b) ((L I) (S P))
2c) (((L) (I) (S) (P)))
2d) (L (I) ((S)) (((P))))
2e) ((((L))) ((I)) (S) P)
2f) ((((L) I) S) P)

Zusätzlich zu den Funktionen der Car-Cdr-Familie gibt es noch die Zugriffsfunktionen First, Second, Tird, Fourth, Fifth, Sixth, Seventh, Eighth, Ninth und Tenth, die eine Liste als Argument erhalten und das erste, zweite, dritte, ..., zehnte Element der Liste als Ergebnis liefern. Die Funktion Nth verlangt eine nichtnegative ganze Zahl n und eine Liste als Argument. Nth liefert das n-te Element der Liste, wobei dem Car der Liste das nullte Element mit Nth entspricht.

Beispiele:
(nth 4 '(a b c d e f)) ⇒ E
(nth 0 '(a b c d)) ⇒ A

Die mehrmalige Anwendung der Funktion Cdr auf eine Liste wird durch Nthcdr erreicht. Wie Nth erwartet Nthcdr eine nichtnegative ganze Zahl und eine Liste als Argumente:

(nthcdr 4 '(a b c d e f))
⇒ (E F)
(nthcdr 28 '(L I S T E))
⇒ NIL

Durch Verschachtelung von mehreren Cons-Aufrufen können entsprechend lange Listen aufgebaut werden. Solche Listen lassen sich aber einfacher durch die Funktionen List bzw. List* erzeugen. Die Funktion List akzeptiert beliebig viele Argumente, die zu einer Liste zusammengefaßt werden.

(list) ⇒ NIL
(list 1 2 3)
== (cons 1 (cons 2 (cons 3 nil)))
⇒ (1 2 3)
(list (list 'a 1) (list 'b 2))
⇒ ((A 1) (B 2))

Die Funktion List* benötigt mindestens ein Argument, wobei das letzte Argument eine Liste sein muß. Als Ergebnis wird eine Liste konstruiert, die aus den ersten (n-1) Argumenten und den Elementen der Liste des letzten Arguments besteht.

(list* 1 2 '(3 4 5))
== (cons 1 (cons 2 '(3 4 5)))
⇒ (1 2 3 4 5)
(list* nil) ⇒ nil
(list* 'a 'b 'c nil) ⇒ (A B C)

Die Funktion Append ist ebenfalls eine Funktion zur Konstruktion von Listen. Sie erwartet zwei Listen als Argumente und bildet aus den Elementen dieser Listen eine neue Liste.

(append '(a b c) '(d e f))
⇒ (A B C D E F)
(append nil '((b) (c))) ⇒ ((B) (C))
(append '(radius) 2.7) nil)
⇒ (RADIUS 2.7)

Die einstellige Funktion Length liefert die Länge einer Liste, das heißt die Anzahl der Listenelemente des obersten Listenniveaus.

(length '(a b)) ⇒ 2
(length '((a b c d e f)) ⇒ 1
(length nil) ⇒ 0

Mit der Funktion Reverse können die Listenelemente auf dem obersten Niveau in ihrer Reihenfolge vertauscht werden:

(reverse '(a b c)) ⇒ (C B A)
(reverse '((a b) (c d)) ⇒ ((C D) (A B))
(reverse nil) ⇒ nil

Sowohl Length als auch Reverse betrachten das ihnen übergebene Argument als eine Liste von Elementen, unabhängig davon, ob die Elemente wiederum Listen oder Atome sind.

Arithmetische Funktionen

Neben den Funktionen für die Grundrechenarten stehen in Lisp traszendente Funktionen, Funktionen für logische Operationen mit ganzen Zahlen, für Operationen mit Teilen ganzer Zahlen (Bytes) und für die Erzeugung von Zufallszahlen bereit.

Die Grundfunktionen +, –, * und / sowie die Funktionen Max (Maximum von n Zahlen), Min (Minimum von n Zahlen), Gcd (größter gemeinsamer Teiler) und Lcm (kleinstes gemeinsames Vielfaches) können beliebig viele Argumente erhalten. Alle arithmetischen Funktionen arbeiten für sämtliche Zahlentypen, man sagt, sie sind *generisch*. Beispiele für arithmetische Funktionsaufrufe:

(+ 2 3 5.5) ⇒ 10.5
(– –23.8) ⇒ 23.8
(* 9 3.0) ⇒ 27.0
(/ 27 3) ⇒ 9
(max 2 4 7 3) ⇒ 7
(min 2 4 7 3) ⇒ 2
(gcd 63 –42 35) ⇒ 7
(lcm 14 35) ⇒ 70
(mod 13 4) ⇒ 1 ; 13 modula 4
(ganzzahliger Rest der Division)
(abs –26.5) ⇒ 26.5 ;
der absolute Betrag von –26.5
(abs 5) ⇒ 5
(expt 2 3) ⇒ 8 ; 2^3
(sqrt 4.0) ⇒ 2.0 ; Quadratwurzel
(random 10) ⇒ 7 ; Zufallszahl
(Random n) liefert eine Zufallszahl, die größer oder gleich Null, aber kleiner als n ist. Der arithmetische Ausdruck

$$\frac{-7+\sqrt{7^2-4\cdot 2\cdot 5}}{2\cdot 2}$$

sieht in der Präfix-Notation von Lisp so aus:
(/ (+ –7 (sqrt (– (expt 7 2) (* 4 2 5)))) (* 2 2))

Übung 3

a) Berechnen Sie folgende Ausdrücke mit Lisp:
2 – 4 + 6 + 8
1 + 15 – 7.0 + 23 – 10.0 – 22
123456789 + 987654321
3.8 + 7.2 – 23.75 + 53.219

b) Erproben Sie die Fehlerreaktion von Lisp, wenn eine Funktion mit zu vielen, mit zu wenigen oder mit falschen Argumenten aufgerufen wird. Versuchen Sie es zum Beispiel mit (car nil '(eine Liste)), (cons 23)) und (+ 34 '(a b c)).

Die Definition von Funktionen

Bestimmte Symbole (z. B. + oder Cons) können auch als Funktionen verwendet werden. Diese Eigenschaft resultiert aus der Zuordnung zu einer Vorschrift, die dann abgearbeitet wird, wenn das Symbol in *funktionaler Stellung* (als erstes Element in einer Liste) auftritt. Diese Eigenschaft wird auch als die funktionale Eigenschaft des Symbols bezeichnet. Der Nutzer kann mit Hilfe der Form **(defun** <Name> <Parameter> <Körper>**)** selbst Funktionen definieren, kann also dem durch <Name> angegebenen Symbol eine funktionale Eigenschaft zuordnen, die durch die Parameter und den Körper spezifiziert wird. Defun ist eine Spezialform, die ihre Argumente nicht auswertet. Das Ergebnis von Defun ist der Name der definierten Funktion. Die Parameter werden durch eine Liste von Symbolen beschrieben, die die formalen Argumente der Funktion darstellen. Beim Aufruf der selbstdefinierten Funktion werden die aktuellen Argumente ausgewertet und der Reihe nach an die Symbole der Parameterliste gebunden. Durch Notation dieser Symbole innerhalb des Körpers kann auf diese Bindung Bezug genommen werden.

(defun zweites (L)
; Ermitteln des zweiten Elementes einer Liste
(car (cdr L)))

Beim Aufruf von (Zweites '(A B C D)) wird nach Auswertung des Arguments die Liste (A B C D) an das Symbol L gebunden. Mit dieser Bindung wird anschließend der Funktionskörper (Car (Cdr L)) abgearbeitet, wobei die Auswertung von L das Ergebnis (A B C D) liefert. Nach Abarbeitung des Funktionskörpers wird die Bindung wieder rückgängig gemacht und das Ergebnis B als Gesamtergebnis des Funktionsaufrufes zurückgegeben.

Übung 4

4a) Definieren Sie je eine Funktion zum Umrechnen von Grad Fahrenheit in Grad Celsius und umgekehrt nach folgenden Formeln:
C = (F+40)/1.8–40 und
F = (C+40)*1.8–40.

4b) Definieren Sie eine Funktion Swap, die als Argument eine zweielementige Liste erhält und diese umdreht. Verwenden Sie dazu nur Car-Cdr-Funktionen und Cons. *Beispiel:*
(swap '(a b)) ⇒ (b a), nicht (b . a)!

4c) Überwachen Sie das Input-Output-Verhalten der Funktion Swap durch Aufruf der Form (trace swap). Rufen Sie anschließend mehrmals Swap auf. Trace ist eine Spezialform, der mehrere Funktionsnamen übergeben werden können. Mit (untrace swap) oder (clear swap) kann das Überwachen wieder ausgeschaltet werden.

Aufgabe 1

Definieren Sie eine Funktion Rotate-left, die eine beliebig lange Liste als Argument erhält. Das Ergebnis soll eine neue Liste sein, bei der das erste Argument der Ausgangsliste das letzte Element der Ergebnisliste ist. Beispiele:
(rotate-left ' (a b c d e f g))
⇒ (b c d e f g a)
(rotate-left (rotate-left '(a b c)))
⇒ (c a b)

Aufgabe 2

Definieren Sie zu der Funktion Rotate-left die Umkehrfunktion Rotate-right.

Prädikate und Konditionale

Um komplizierte Funktionen schreiben zu können, werden Prädikate benötigt. Prädikate liefern als Ergebnis immer die Aussage *wahr* oder *falsch*. In Lisp wird für *falsch* das Symbol NIL verwendet, und jeder Wert ungleich NIL bedeutet *wahr*, also auch das Symbol T, das im Normalfall dafür benutzt wird.

Typprädikate

Typprädikate sind die wichtigsten und auch die am häufigsten verwendeten Prädikate in Lisp. Sie geben Auskunft darüber, ob ein Objekt von einem bestimmten Typ ist. Wir kennen bis jetzt Atome, Zahlen, Symbole, Zeichenketten und Listen. Um nichtleere Listen von Atomen unterscheiden zu können, gibt es den Datentyp Cons. Das Symbol NIL ist das einzige Element des Datentyps Null, der damit nicht nur Teiltyp von Symbol, sondern auch von List ist. Bild 1 zeigt, wie unser Typbaum bis jetzt aussieht.

Die jeweiligen Typprädikate heißen Atom, Numberp, Fixnump, Floatp, Stringp, Symbolp, Listp, Null und Consp. Ihre Wirkungsweise erkennt man am besten an den folgenden Beispielen:

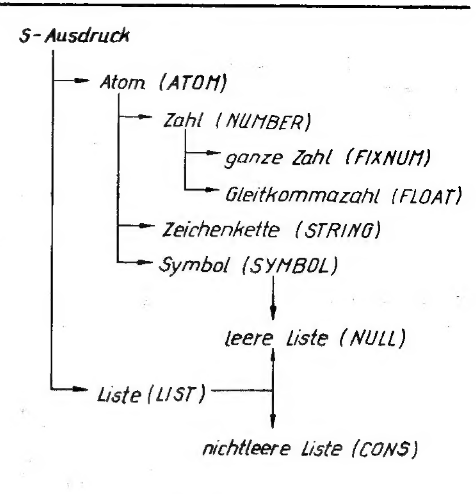


Bild 1 Die Typhierarchie in Lisp

(atom nil) ⇒ T
(atom 'a) ⇒ T
(atom '(a b c)) ⇒ NIL
(numberp 33) ⇒ T
(numberp 27.86) ⇒ T
(numberp 'five) ⇒ NIL
(fixnump 24) ⇒ T
(floatp 24) ⇒ NIL
(floatp 24.0) ⇒ T
(stringp "22.3") ⇒ T
(stringp 'string) ⇒ NIL
(listp '(a b c)) ⇒ T
(listp 'symbol) ⇒ NIL
(listp nil) = = (listp ()) ⇒ T
(consp '(1 2 3)) ⇒ T
(consp NIL) ⇒ NIL
(null nil) ⇒ T

Alle Typprädikate lassen sich auch durch das Prädikat Typep realisieren:
(typep <Objekt> <Typ>**)**
liefert T, wenn das Argument <Objekt> vom Typ <Typ> ist, ansonsten NIL.

(typep 3 'number) ⇒ T
(typep 3 'fixnum) ⇒ T
(typep 3 'float) ⇒ NIL

Vergleichsprädikate

In Lisp gibt es verschiedene Gleichheitsbegriffe und daher auch verschiedene Prädikate zum Testen auf Gleichheit. Der einfachste Test ist der Test auf physische Gleichheit (Eq). Eq liefert T, wenn die beiden Argumente physisch gleich sind:
(eq 'g 'g) ⇒ T
(eq '(a b c) '(a b c)) ⇒ NIL
Will man für zwei unterschiedliche Listen prüfen, ob sie die gleichen Elemente besitzen, so muß man Equal verwenden.
(equal '(1 2 3) '(1 2 3)) ⇒ T, aber
(eq '(1 2 3) '(1 2 3) ⇒ NIL,
denn bei jedem Einlesen von (1 2 3) wird physisch eine neue Liste erzeugt.
Zahlen werden mit den Funktionen =, >, <, <= und >= verglichen, wobei diese beliebig viele Argumente akzeptieren. Beispielsweise liefert das Prädikat <= als Ergebnis *wahr*, wenn die Folge der übergebenen Argumente eine monoton steigende Zahlenfolge ist. Mit

diesen Funktionen lassen sich sehr einfach Intervalltests formulieren:

(>= 9 8 7 7 5 4 2) ⇒ T
(> 9 8 7 7 5 4 2) ⇒ NIL
(= 3 3.0) ⇒ T, während aber gilt
(equal 3 3.0) ⇒ NIL.

Zum Vergleich von Zeichenketten werden die Funktionen *String=*, *String>*, *String<*, *String<=*, *String>=* und *String/=* verwendet. Diese zweistelligen Funktionen können auch auf Symbole angewendet werden:

(string< "abc" "abd") ⇒ T
(string= "foo" "FOO" ⇒ NIL
(string= "LISP" 'lisp) ⇒ T

Der letzte Vergleich liefert T, weil Lisp alle Buchstaben in Symbolen zu Großbuchstaben umwandelt. Die Funktionen String-equal, String-greaterp, String-lessp, String-not-greaterp, String-not-lessp und String-not-equal ignorieren beim Vergleich die Unterschiede zwischen Groß- und Kleinbuchstaben.

Logische Verknüpfungen

Mehrere Prädikate lassen sich aussagenlogisch durch
(and <wert-1> ... <wert-n>),
(or <wert-1> ... <wert-n>) *und*
(not <wert>)
verknüpfen. Die Funktion *Not* und die Spezialformen And und Or arbeiten so, wie man es von der Aussagenlogik her gewohnt ist. And wertet seine Argumente nur soweit aus, bis es auf eines trifft, das NIL liefert; die restlichen Argumente werden nicht beachtet. Or hingegen wertet nur solange aus, bis es auf ein Argument trifft, das einen Wert ungleich NIL liefert. Zum Beispiel werden bei
(and t t nil) ⇒ NIL
nacheinander alle Argumente ausgewertet, wobei
(or t t nil) ⇒ T
nur das erste Argument auswertet. Die Funktion *Not* arbeitet wie Null und liefert als Ergebnis T, wenn sie als Argument NIL erhielt.

```
(defun constantp (x)
    (or (null x) (eq x t)
        (numberp x) (stringp x)
        (and (consp x)
            (eq (car x) 'quote))))
```

Programmverzweigungen

Die Spezialform If stellt die einfachste Möglichkeit dar, Programmverzweigungen zu realisieren. If ist mit den If-then-else-Konstruktionen algebraischer Programmiersprachen vergleichbar und hat folgende Syntax:
(if <test> **<then>** [**<else>**]**)**
If wertet zuerst die Form *Test* aus. Ist das Ergebnis ungleich NIL, wird die *Then*-Form ausgewählt, sonst die *Else*-Form. Diese ausgewählte Form wird anschließend ausgewertet und liefert das Gesamtergebnis des If-Aufrufes. Wurde die Else-Form weggelassen, so wird dafür NIL genommen.

(if (numberp 3829) 'ja 'nein) ⇒ JA
(if (consp 'a) 'cons 'kein-cons)
⇒ KEIN-CONS
(if (listp 392) 'liste) = =
(if (listp 392) 'liste nil) ⇒ NIL

Da eine Verschachtelung vieler If-Formen schwer lesbar ist, gibt es die Spezialform Cond – ähnlich den Swith- oder Case-Anweisungen anderer Programmiersprachen – mit der folgenden Syntax:
(cond (<prädikatform-1> <form-11>
... <form-1m>)
(<prädikatform-n> <form-n1>
... <form-nk>**))**
Die erste Prädikatform wird ausgewertet. Liefert sie einen Wert ungleich NIL, so werden alle nachfolgenden Formen <form-11> bis <form-1m> ausgewertet und die Abarbeitung von Cond wird abgebrochen, ansonsten wird zur zweiten Prädikatform übergegangen usw. Als letzte Prädikatform empfiehlt es sich, T anzugeben. Da dies immer ungleich NIL ist, wird die Abarbeitung der letzten Alternative gesichert, wenn alle anderen verworfen werden mußten. Als Beispiel dafür wird die Funktion Typep definiert:

```
(defun typep (objekt typ)
    (cond ((eq typ 'cons) (consp objekt))
        ((eq typ 'list) (listp objekt))
        ((eq typ 'symbol) (symbolp objekt))
        ((eq typ 'atom) (atom objekt))
        (t "unbekannter Datentyp")))
```

Aufgabe 3
Definieren Sie eine Funktion SIMP+, die einen arithmetischen Ausdruck vereinfacht, wobei nur die Regeln der Addition berücksichtigt werden (z. B. 0 + n = n). Als Argument erhält die Funktion eine Liste, die aus dem Element +, dem linken Operanden und dem rechten Operanden besteht. Als Operanden können Zahlen und Symbole verwendet werden.

(simp+ '(+ 27.4 63.2)) ⇒ 100.6
(simp+ '(+ 0 b)) ⇒ B
(simp+ '(+ a c)) ⇒ (+ a c)

Im letzten Beispiel ist keine Vereinfachung möglich.

Literatur
Charniak, E.; Riesbeck, C. K.; McDermott, D. V.; Meehan, J. R.: Artifical Intelligence Programming, Lawrence Erlbaum Associates, Hillsdate, New Jersy 1987
Steele, G. L.: Common LISP: The Language, Digital Press 1984.
Stoyan, H.: Programmiermethoden der Künstlichen Intelligenz. Band 1, Studienreihe Informatik, Springer-Verlag, Berlin, Heidelberg, New York, London, Paris, Tokio 1988
Winston P. H., Horn B. K. P.: LISP, Addison-Wesley Publishing Company, Reading (Mass.) 1988

```
1)  (DEFUN ROTATE-LEFT (LISTE)
          (APPEND (CDR LISTE)
                  (LIST (CAR LISTE))))

2a) (DEFUN ROTATE-RIGHT1 (LISTE)
          (REVERSE
             (ROTATE-LEFT (REVERSE LISTE))))

2b) (DEFUN ROTATE-RIGHT2 (LISTE)
          (APPEND
             (NTHCDR (- (LENGTH LISTE) 1)
                     LISTE)
             (REVERSE
                (CDR (REVERSE LISTE)))))

3)  (DEFUN SIMP+ (FORMEL)
       (COND ((AND (NUMBERP (SECOND FORMEL))
                   (NUMBERP (THIRD FORMEL)))
              (+ (SECOND FORMEL)
                 (THIRD FORMEL)))
             ((AND (NUMBERP (SECOND FORMEL))
                   (ZEROP (SECOND FORMEL)))
              (THIRD FORMEL))
             ((AND (NUMBERP (THIRD FORMEL))
                   (ZEROP (THIRD FORMEL)))
              (SECOND FORMEL))
             ((EQ (SECOND FORMEL)
                  (THIRD FORMEL))
              (LIST '* 2 (SECOND FORMEL)))
             (T FORMEL)))
```

Bild 2 Lösungen der Aufgaben 1 bis 3

Zur Lösung der Aufgabe 3 sei folgendes angemerkt: Die Funktion besteht aus einem Cond, in dessen Zweigen die Bedingungen für eventuelle Vereinfachungen geprüft werden. Der erste Zweig sorgt für die Ausführung der Addition, wenn beide Summanden Konstanten sind. Der zweite und der dritte Zweig sorgen dafür, daß keine unnötige Addition mit 0 durchgeführt wird. Der vierte Zweig wandelt die Addition zweier gleicher Summanden in eine Multiplikation um. Der fünfte Zweig hat als Bedingung T, das heißt, in jedem anderen Fall ist keine Vereinfachung möglich.

✉ KONTAKT ☎

Akademie der Wissenschaften der DDR, Zentralinstitut für Kybernetik und Informationsprozesse, Kurstraße 33, Berlin, 1086; Tel. 20376

TERMINE

Forth-Lehrgänge

WER? Kammer der Technik Suhl
WANN?
1. Lehrgang: 3. bis 7. 9. 1990
2. Lehrgang: 10. bis 14. 9. 1990
WO? Suhl
WAS?
- Möglichkeiten von Forth-83
- Programmieren selbständig ausgewählter Probleme
- Überblick über verschiedene Forth-Prozessoren

WIE Auskünfte und Anmeldungen bitte bei Kammer der Technik Suhl, Bereich Weiterbildung, Postfach 510, Suhl, 6000; Tel. 22112 (Kolln. Keller)
Dr. Finsterbusch

1. Problemseminar „Programmierung von 32-Bit-Minicomputersystemen"

WER? Kammer der Technik, Fachverband Elektrotechnik, Fachausschuß Minicomputersysteme und DECUS München e. V.
WANN? 17. bis 21. September 1990
WO? Cottbus
WAS?
- Betriebssysteme und Standardsoftware für Minicomputersysteme wie VAX-11, K 1840, K 1820, CM 52/12, Elektronika-82, CM 1700
- Generierung, Management, Optimierung, Anwendungsprobleme und Anwendungserfahrungen

WIE? Teilnahmemeldungen richten Sie bitte an: Kammer der Technik, Bezirksvorstand Cottbus, Stadtpromenade 3, Postschließfach 90/1 Cottbus, 7500; Tel. 24095
Prof. Dr. Horn

Lisp

Teil 2

Dr. Ulrich Kriegel

Rekursionen

Programmierung basiert auf der Auswahl von Kontrollstrukturen, die festlegen, wie die einzelnen Programmschritte ausgeführt werden sollen. Ein typisches Beispiel für eine derartige Kontrollstruktur in Lisp ist die Rekursion. Der Grundgedanke dabei ist, daß eine Funktion ein Problem etwas vereinfacht und dieses dann an eine oder mehrere identische Kopien von sich selbst zur weiteren Verarbeitung übergibt. In Abhängigkeit davon, ob bei einem Rekursionsschritt die Funktion ein- oder mehrmals aufgerufen wird, spricht man von Einfach- bzw. Mehrfach-Rekursionen.

Eine rekursive Funktion muß immer die folgenden induktiven Aspekte des zu behandelnden Problems enthalten:
① Es muß ein Test auf eine Abbruchbedingung existieren, bei der die Rekursion abgebrochen wird.
② Es muß eine Methode für die Vereinfachung des Problems existieren, so daß die Folge der vereinfachten Probleme gegen die Abbruchbedingung konvergiert.
Im folgenden Abschnitt sollen die möglichen Formen der Rekursion und ihre Anwendung bei der Synthese und Analyse von S-Ausdrücken behandelt werden.

Rekursive Synthese von S-Ausdrücken

Als Beispiel für eine einfach-rekursive Funktion soll die in Bild 1 gezeigte Funktion MUL dienen, die zwei positive ganze Zahlen A und B multipliziert. Die Funktion bricht ab (terminiert), da B schrittweise um Eins verkleinert und damit einmal der Test (= B O) erfüllt wird. Die Funktion ist korrekt, denn es gilt:
$A * B = A + (B - 1) * A$
Das in Bild 2 gegebene Trace-Protokoll zeigt die Abarbeitung von (MUL 4 2). Da auf jedem Rekursionsniveau die Funktion MUL nur einmal aufgerufen wird, ist der Aufrufbaum linear. Aus den zurückgegebenen Funktionswerten ist ersichtlich, daß das Gesamtergebnis aus den Teilergebnissen zusammengesetzt wird. Man spricht deshalb auch von rekursiver Synthese.
Ein Beispiel für eine doppelt-rekursive Funktion soll anhand der Berechnung der sogenannten Fibonacci-Zahlen gegeben werden. Leonardo von Pisa, auch Fibonacci genannt, stellte ein diskretes Modell für das Wachstum einer Population von Hasen auf, das durch die folgende Gleichung beschrieben wird:
$f(n) = f(n - 1) + f(n - 2)$
wenn $n > 1$ ist, sonst gilt $f(n) = 1$.
Bild 3 zeigt die Funktion Fibonacci, Bild 4 das Trace-Protokoll des Aufrufs von (Fibonacci 4). Der Aufrufbaum ist nicht mehr linear wie im vorhergehenden Beispiel, da auf jeder Rekursionsebene zwei Kopien der Funktion Fibonacci aufgerufen werden.
Nach der Lösung dieses relativ einfachen numerischen Problems wollen wir uns nun wieder der Symbolverarbeitung zuwenden. Ein weiteres Beispiel für die rekursive Synthese von S-Ausdrücken ist eine Funktion, die Listen auf dem obersten Niveau kopiert. Da die in Common Lisp definierte Funktion Copy-List etwas allgemeiner anwendbar ist als die folgende Definition, wollen wir sie MP-copy-lisp nennen (Bild 5). Allgemein wird in diesem Kurs immer dann ein Funktionsname mit dem Präfix MP versehen, wenn eine Funktion gleichen Namens im Common-Lisp-Standard existiert, aber die hier vorgestellte Funktion im Vergleich zum Standard nur eingeschränkt verwendbar ist. Die Auswertung von (MP-copy-list '(A B C)) liefert das in Bild 6 dargestellte Aufrufprotokoll.

```
(defun mul
       (a b)
  (cond ;; b = 0 ?
       ((= b 0)0)
       ;; Rekursion
       (t (+ a (mul a (- b 1))))))
```

Bild 1 Definition der Funktion Mul

```
-> MUL (4 2)
I-> MUL (4 1)
II-> MUL (4 0)
II<- MUL (0)
I<- MUL (4)
<- MUL (8)
```

Bild 2 Trace-Protokoll des Aufrufs (Mul 4 2)

```
(defun fibonacci
      (n)
  (cond ;; Abbruch n <= 1
       ((>= 1 n) 1)
       ;; doppelte Rekursion
       (t (+ (fibonacci (- n 1))
             (fibonacci (- n 2))))))
```

Bild 3 Definition der Funktion Fibonacci

```
-> (FIBONACCI 4)
I-> (FIBONACCI 3)
II-> (FIBONACCI 2)
III-> (FIBONACCI 1)
III<- (FIBONACCI 1)
III-> (FIBONACCI 0)
III<- (FIBONACCI 1)
II<- (FIBONACCI 2)
II-> (FIBONACCI 1)
II<- (FIBONACCI 1)
I<- (FIBONACCI 3)
I-> (FIBONACCI 2)
II-> (FIBONACCI 1)
II<- (FIBONACCI 1)
II-> (FIBONACCI 0)
II<- (FIBONACCI 1)
I<- (FIBONACCI 2)
<- (FIBONACCI 5)
```

Bild 4 Rease-Protokoll des Aufrufs (Fibonacci 4)

```
(defun mp-copy-list
       (li)
  (cond ((null li) nil)
        (t (cons (car li)
              (mp-copy-list (cdr li)))))).
```

Bild 5 Definition der Funktion MP-copy-list

```
-> MP-COPY-LIST ((A B C))
I-> MP-COPY-LIST ((B C))
II-> MP-COPY-LIST ((C))
III-> MP-COPY-LIST (NIL)
III<- MP-COPY-LIST (NIL)
II<- MP-COPY-LIST ((C))
I<- MP-COPY-LIST ((B C))
<- MP-COPY-LIST ((A B C))
```

Bild 6 Trace-Protokoll des Aufrufs (mp-copy-list '(A B C))

Aufgabe 4: Schreiben Sie eine Funktion, die rekursiv die Länge einer Liste bestimmt. (Die Lösungen der Aufgaben dieses Teils finden Sie ausnahmsweise erst im Teil 3 in MP 11/90.)

```
(defun mp-member
       (item li)
  (cond ;;Liste leer ?
       ((null li) nil)
       ;;ITEM gefunden ?
       ((equal item  (car li)) li)
       ;;Rekursion
       (t(mp-member item (cdr li)))))
```

Bild 7 Definition der Funktion MP-member

Restrekursion – die rekursive Analyse von S-Ausdrücken

Ebenso wie zur Synthese können Rekursionen auch zur Analyse von S-Ausdrücken benutzt werden.
Nehmen wir einmal an, daß in unserem Lisp die Grundfunktion Member des Lisp-Standards nicht vorhanden ist. Member testet, ob das erste Argument auf dem obersten Niveau in der Liste enthalten ist, die als zweites Argument übergeben wird (Bild 7). MP-Member behandelt zwei Basissituationen: Entweder ist die Liste leer, dann wird NIL zurückgegeben, oder ITEM ist das erste Element der Liste LI, dann wird LI zurückgegeben. Tritt keine dieser Basissituationen auf, so übergibt MP-Member ein vereinfachtes Problem, nämlich die Suche von ITEM in der um ein Element verkürzten Liste, an eine Kopie von sich selbst. In Bild 8 ist die Abarbeitung des Aufrufs von (MP-Member 'C '(A B C D E F)) dargestellt. Beachten Sie bitte, daß MP-Member entweder eine Basissituation erkennt oder das Problem *delegiert*. Im Gegensatz zu MP-copy-list müssen deshalb auch keine Zwischenergebnisse gespeichert werden, während die Kopie arbeitet. Der Wert, den MP-Member zurückgibt, ist genau der Wert, den die Kopie auf dem untersten Niveau der Rekursion liefert.

```
-> MP-MEMBER (C (A B C D E F))
I-> MP-MEMBER (C (B C D E F))
II-> MP-MEMBER (C (C D E F))
II<- MP-MEMBER ((C D E F))
I<- MP-MEMBER ((C D E F))
<- MP-MEMBER ((C D E F))
```

Bild 8 Trace-Protokoll des Aufrufs (MP-member 'C '(A B C D E F))

Allgemein nennt man eine Funktion *restrekursiv*, wenn der zurückgegebene Wert entweder direkt berechnet wurde oder (wie oben) der Wert des rekursiven Aufrufs auf dem untersten Rekursionsniveau ist. Restrekursive Funktionen müssen den Wert, den die rekursiven Aufrufe liefern, nicht weiter bearbeiten. Vom Prinzip her muß deshalb die Kopie auch nicht zu der sie aufrufenden Kopie (bzw. zum Original) zurückkehren; statt dessen kann direkt zu der rufenden Funktion verzweigt werden. Optimierende Compiler, z. B. der XCL-Compiler, nutzen diese Tatsache aus und erzeugen für restrekursive Funktionen äußerst effektiven Maschinenkode. Das ist auch der Grund, warum von vielen Programmierern der Programmierstil mit restrekursiven Funktionen bevorzugt wird. Mit einem Trick ist es sogar möglich, eine bestimmte Klasse nicht restrekursiver Funktionen in restrekursive Funktionen umzuwandeln.

Restrekursion mit Akkumulatorvariable

Bild 9 zeigt die Definition der Funktion Copy-N, die aus einer Liste die Teilliste der maximal ersten n Elemente erzeugt. Die Funktion ist nicht restrekursiv, weil sie das Ergebnis des Aufrufs der Kopie der Funktion Cons übergibt, um die Ergebnisliste zu bilden.
Rufen wir, wie in Bild 10 dargestellt, statt Copy-n eine Hilfsfunktion rekursiv auf, so kann diese restrekursiv programmiert werden. Der Trick besteht darin, daß die Teilergebnisse des jeweiligen Rekursionsniveaus dem nächsten Niveau mit übergeben werden. Die Hilfsfunktion ist restrekursiv, weil der Wert des rekursiven Aufrufs in Cond direkt als Rückgabewert benutzt wird.
Generell ist eine Restrekursion mit Akkumulatorvariable immer dann möglich, wenn bei jeder Rekursion ein Teilergebnis bestimmbar ist. Diese Tatsache nutzen optimierende Compiler aus. Bei der Kode-Optimierung werden dann automatisch restrekursive Hilfsfunktionen generiert.

Aufgabe 5: Definieren Sie die Funktion MP-reverse, die die oberste Ebene einer Liste umdreht, als restrekursive Funktion mit Akkumulatorvarable.

Rückwärtsrekursion – die Synthese einer Liste vom Ende her

Zur Lösung mancher Probleme ist es oft günstig, eine Liste vom Ende her abzubauen. Die naheliegenste Lösung hierbei wäre wohl, die Liste erst mit Reverse umzudrehen und dann zu bearbeiten. Bei langen Listen ist dieses Verfahren jedoch zeit- und speicherplatzaufwendig. Einen Ausweg bietet hier die Rückwärtsrekursion, die im folgenden am Beispiel der Funktion Copy-last erläutert werden soll (Bild 11). Copy-last kopiert die letzten n Elemente einer Liste.
Zum Verständnis der Arbeitsweise rückwärtsrekursiver Funktionen ist es jedoch notwendig, sich etwas eingehender mit dem Gültigkeitsbereich von Variablen in Common Lisp zu beschäftigen. Normalerweise hat in Common Lisp eine Variable, die in der Parameterliste einer Funktion auftritt oder in der Funktion als Hilfsvariable spezifiziert wird, einen lexikalischen Gültigkeitsbereich; das bedeutet, daß sie innerhalb der Funktion frei verfügbar ist, ihre Bindung aber außerhalb der Funktion unbestimmt ist. Den Gegensatz dazu bilden dynamisch gebundene Variablen (auch Special-Variablen genannt), die einen dynamischen Gültigkeitsbereich besitzen, das heißt, sie existieren so lange, wie das sie etablierende Konstrukt existiert. Wird nun die Variable **n** innerhalb der Funktion Copy-last durch **(declare (special n))** als dynamisch gebundene Variable deklariert, so können alle Funktionen, die von dieser Funktion aufgerufen werden – in diesem Falle alle Kopien der Hilfsfunktion Copy-last-aux, auf diese Variable zugreifen. Diese Tatsache wollen wir bei der Ablaufsteuerung ausnutzen: Die Hilfsfunktion soll die Liste rekursiv abbauen und dann vom Ende beginnend, das Ergebnis synthetisieren. Dazu werden die Ergebnisse der einzelnen Kopien der Hilfsfunktionen in einer lokalen Variablen L1 abgelegt. Die mit dem letzten Element der Liste aufgerufene Kopie der Hilfsfunktion verringert (dekrementiert) n um Eins und gibt das letzte Element zurück, das an die Hilfsvariable L1 der übergeordneten Kopie gebunden ist. Daraus und aus dem ersten Element der Input-Liste wird eine neue Teilliste erzeugt, und n wird dekrementiert. Der Prozeß bricht ab, wenn n gleich Null ist. (In unvollständigen Implementationen von Common Lisp, beispielsweise in den ersten Versionen von GC-Lisp, waren nur dynamische Variablenbindungen möglich. In diesem Fall muß die Deklaration als dynamische Variable entfallen.)
In der Definition Copy-last-aux ist die Spezialform Let zur Deklaration lokaler Variable enthalten, die die folgende Syntax besitzt.

(LET ((VAR-1 INIT-1) . . . (VAR-n INIT-n))
(ANWEISUNG-1)
. . .
(ANWEISUNG-n))

VAR-1 bis VAR-n sind im Normalfall lokale lexikal gebundene Variablen, die durch die optionalen Formen Init-1 bis Init-n initialisiert werden. Die Initialisierung erfolgt parallel, das heißt, zuerst werden alle Initialisierungsformen ausgewertet, und danach erfolgt die Initialisierung. Werden die Variablen ohne Initialisierungsformen angegeben, so müssen die Variablennamen nicht geklammert werden. Nach der Liste (...(VAR-i INIT-i)...) bzw. (...VARi...) kann optional eine Deklaration folgen, in der beispielsweise einige der Variablen als dynamisch gebundene Variable spezifiziert werden. Danach folgt der Körper der Let-Form, das heißt, beliebig viele Anweisungen, die sequentiell abgearbeitet werden. Der Gültigkeitsbereich der lexikalen Var-i ist auf den Körper der Let-Form beschränkt. Das Ergebnis der letzten Anweisung im Körper ist gleichzeitig das Ergebnis der Let-Form. Neben der Spezialform Let existiert noch die Spezialform Let* mit analoger Syntax, die jedoch im Gegensatz zu Let die Variablen sequentiell initialisiert.
Zur Veranschaulichung der Arbeitsweise von Copy-last-aux enthält Bild 12 ein Trace-Protokoll mit den zusätzlich eingetragenen Werten von n und L1.

```
(defun copy-n
       (li n)
 (cond ;;fertig ?
       ((= n 0) nil)
       ;;Rekursion
       (t(cons (car li)
             (copy-n (cdr-li)(- n 1))))))
```

Bild 9 Definition der Funktion Copy-n

```
(defun copy-n
       (li n)
 (copy-n-aux li n nil))

(defun copy-n-aux
       (li n acc)
 (cond ;;fertig ? -> umdrehen der Liste
       ;;im Akkumulator
       ((= n 0)(reverse acc))
       ;Restrekursion
       (t (copy-n-aux (cdr li)
              (- n 1)
              (cons (car li) acc)))))
```

Bild 10 Definition der Funktion Copy-n mit restrekursiver Hilfsfunktion

```
(defun copy-last
       (li n)
 (declare (special n)) ;Deklariere n
 (copy-last-aux li))   ;Aufruf der Hilfsfkt.

(defun copy-last-aux
       (l)
 (cond ((null l) nil)
       (t (let ((l1 (copy-last-aux
                     (cdr l))))
            (cond ((= 0 n) l1)
                  (t (setq n (- n 1))
                     (cons (car l)
                       l1))))))
```

Bild 11 Definition der Funktion Copy-last

```
->  COPY-LAST-AUX ((A B C D))    N = 2  L1 = ?
I-> COPY-LAST-AUX ((B C D))      N = 2  L1 = ?
II-> COPY-LAST-AUX ((C D))       N = 2  L1 = ?
III-> COPY-LAST-AUX ((D))        N = 2  L1 = ?
IIII-> COPY-LAST-AUX (NIL)       N = 2  L1 = ?
IIII<- COPY-LAST-AUX (NIL)       N = 2  L1 = ?
III<- COPY-LAST-AUX ((D))        N = 2  L1 = NIL
II<- COPY-LAST-AUX ((C D))       N = 1  L1 = (D)
I<- COPY-LAST-AUX ((C D))        N = 0  L1 = (C D)
<- COPY-LAST-AUX((C D))          N = 0  L1 = (C D)
```

Bild 12 Trace-Protokoll des Aufrufs (copy-last-aux) '(A B C D))

Aufgabe 6: Schreiben Sie die Funktion MP-reverse, die die Elemente einer Liste umdreht, unter Ausnutzung der Rückwärtsrekursion.

Rekursive Analyse und Synthese

In den vorangegangenen Abschnitten haben wir Funktionen zur rekursiven Analyse und zur rekursiven Synthese symbolischer Ausdrücke getrennt behandelt. In der Programmierpraxis tritt jedoch häufig der Fall auf, daß rekursive Synthese und Analyse gekoppelt sind. Als Beispiel soll die Funktion MP-count dienen, die die Anzahl der Atome in einem S-Ausdruck feststellt (Bild 13). Die Funktion be-

```
(defun mp-count
       (l)
  (cond ;;Liste leer ?
        ((null l) 0)
        ;;atom ?
        ((atom l) 1)
        ;;analysiere car und cdr
        ;;und addiere Ergebnisse
        (t (+ (mp-count (car l))
              (mp-count (cdr l))))))
```

Bild 13 Definition der Funktion MP-count

```
-> (MP-COUNT (* A (SIN B)))
I-> (MP-COUNT *)
I<- (MP-COUNT 1)
I-> (MP-COUNT (A (SIN B)))
II-> (MP-COUNT A)
II<- (MP-COUNT 1)
II-> (MP-COUNT ((SIN B)))
III-> (MP-COUNT (SIN B))
IIII-> (MP-COUNT SIN)
IIII<- (MP-COUNT 1)
IIII-> (MP-COUNT (B))
IIIII-> (MP-COUNT B)
IIIII<- (MP-COUNT 1)
IIIII-> (MP-COUNT NIL)
IIIII<- (MP-COUNT 0)
IIII<- (MP-COUNT 1)
III<- (MP-COUNT 2)
III-> (MP-COUNT NIL)
III<- (MP-COUNT 0)
II<- (MP-COUNT 2)
I<- (MP-COUNT 3)
<- (MP-COUNT 4)
```

Bild 14 Trace-Protokoll des Aufrufs (mp-count '(∗ a (sin b)))

handelt 2 Basisfälle, nämlich die leere Liste und das Atom. Treffen beide Fälle nicht zu, so wird MP-count auf Car und Cdr der Liste getrennt angewendet. Die Ergebnisse der Aufrufe werden dann addiert. Diese Funktion ist doppelt-rekursiv. Zur Illustration enthält Bild 14 das Trace-Protokoll des Aufrufs **(mp-count '(∗ a (sin b)))**.

Allgemein kann gesagt werden, daß einfach-rekursive Funktionen die natürliche Methode sind, um mit Sequenzen, beispielsweise mit den obersten Elementen von List zu arbeiten. Die Verwendung doppelt-rekursiver Funktionen bieten sich dann an, wenn es darum geht, verschachtelte Listen auch in der Tiefe zu untersuchen, so daß dabei alle Elemente erreicht werden. Der Berechnungsaufwand für einfach-rekursive Funktionen ist dabei proportional der Länge der zu bearbeitenden Liste, während er für doppelt-rekursive Funktionen proportional zur Anzahl der Elemente in der Liste ist.

Der Terminplaner

Wir wollen nun mit der Entwicklung von Funktionen beginnen, die es erlauben, einen einfachen Terminplan zu verwalten. Terminpläne für bestimmte Personen sollen aufgestellt, freie Termine ermittelt, Termine in Pläne verschiedener Nutzer eingetragen oder eine Terminliste soll ausgegeben werden. Das Programm ist so angelegt, daß Sie es nach Ihren Wünschen selbst erweitern können. So wäre es beispielsweise denkbar, daß die Kalenderstrukturen in einer externen Datei gesichert werden. Bei jedem Start des Systems könnten sie dann automatisch geladen werden, und in Abhängigkeit vom Tagesdatum wird die Liste der nächsten Termine ausgegeben.

Der Terminplaner weist folgende logische Struktur auf:

• Er besteht aus einer Liste von Tagesplänen.
• Ein Tagesplan soll das Tagesdatum und eine Liste von Terminen enthalten.
• Ein Tagesdatum setzt sich aus dem Wochentag, dem Tag und dem Monat zusammen.
• Ein Termin besteht aus Beginn- und Endzeitpunkt und aus dem Thema.
• Eine Zeitangabe enthält Stunden- und Minutenangaben.

Wir haben es also mit Objekten zu tun, die eine unveränderliche Struktur aufweisen. Nur die Inhalte der einzelnen Datenfelder, die auch Slots genannt werden, können sich jeweils ändern. Wir können derartige Strukturen natürlich mittels Listen darstellen. Auf die einzelnen Slots könnte man dann über die Funktionen der Car/Cdr-Familie zugreifen.

Man kann sich leicht vorstellen, wie gut lesbar dann Programme wären, die mit verschachtelten Strukturen arbeiten. Eine Änderung der Datenstruktur, beispielsweise der Reihenfolge der Slots, hätte schwerwiegende Folgen. Im gesamten Programm müßten beispielsweise Aufrufe von Car durch solche von Cadr ersetzt werden.

Dabei sind Fehler möglich, denn bestimmte Aufrufe von Car könnten auch zur Steuerung der Rekursion dienen.

Ein Ausweg aus dieser Misere besteht in der Definition abstrakter Datentypen, bei denen die physische Repräsentation der Daten dem Nutzer verborgen bleibt. Der Zugriff auf die einzelnen Slots einer solchen abstrakten Datenstruktur erfolgt über sogenannte Selektorfunktionen.

Programme, die abstrakte Datentypen enthalten, sind bedeutend leichter zu verstehen, und vor allen Dingen lassen sie sich leichter warten.

Ist das Datum eine Liste aus Wochentag, Tag und Monat, so könnte eine Selektorfunktion für den Monatsslot wie folgt aussehen:

```
(defun datum-monat(datum)
(caddr datum))
```

Zeiteinbußen durch die zusätzlichen Funktionsaufrufe bei der Verwendung abstrakter Datentypen können vermieden werden, wenn die entsprechenden Funktionen als Inline-Funktionen deklariert werden. Der Compiler kann diese Funktionsaufrufe dann durch den entsprechenden Inline-Kode ersetzen, beispielsweise (datum-monat datum) durch (caddr datum). Die Anzahl der Selektorfunktionen wächst linear mit der Zahl der Slots in der Struktur. Viele Dialekte von Lisp stellen deshalb dem Programmierer mit Defstruct ein Hilfsmittel zur Verfügung, das automatisch die entsprechenden Selektorfunktionen, ein Typprädikat, eine Konstruktorfunktion und eine Kopierfunktion erzeugt.

In Common Lisp hat der Aufruf von Defstruct im einfachsten Falle die folgende Syntax:

```
(defstruct name
    (slot-1 init-1)
    (slot-2 init-2)
    ...
    (slot-n init-n)).
```

Mit diesem Aufruf werden

- die Konstruktorfunktion *Make-name*,
- n Selektorfunktionen *Name-slot-i*,
- eine Kopierfunktion *Copy-name* und
- ein Typprädikat *Name-p*

generiert.

Der Aufruf von **(Make-name)** erzeugt dann eine Struktur mit n Slots, die mit den Werten Init-i initialisiert wurden. Werden keine Init-Werte angegeben, so können die Klammern um die Slotnamen entfallen. Die Slots werden dann mit NIL initialisiert.

Soll bei der Konstruktion der Struktur m Slots explizit ein Wert zugewiesen werden, so geschieht das durch den Aufruf (make-name :slot-i Wert-i). Symbole, die mit einem Doppelpunkt beginnen, werden auch Schlüsselwörter genannt. Diese haben sich selbst zum Wert und müssen deshalb nicht *quotiert* werden. Durch die Angabe der Schlüsselwörter zur Kennzeichnung der entsprechenden Slots ist die Reihenfolge der Schlüsselwort – Wert – Paare beliebig. Die physische Umsetzung der Struktur hängt von der Implementation ab. Nun stellt sich die Frage, wie man die Inhalte der einzelnen Slots manipulieren kann. Müßte es nicht auch dafür n Funktionen geben, für jeden Slot eine?

Um dieses Problem allgemein lösen zu können, wurde in Common Lisp das Konzept der verallgemeinerten Variablen eingeführt. Unter einer verallgemeinerten Variable wird dabei ein beliebiges Lisp-Objekt verstanden, das Daten aufnehmen kann und für das durch eine entsprechende Selektorfunktion ein Zugriffspfad definiert ist. So ist beispielsweise Cadr einer Liste LI eine verallgemeinerte Variable, der Zugriffspfad ist (cadr li). Mit Hilfe von Setf kann eine verallgemeinerte Variable belegt werden:

(**Setf** <zugriffspfad><wert>).

Beispiel:
(setq li '(a b)) ==> (A B)
(setf (cadr li) 'x) ==> X
li ==> (A X).

Da die einzelnen Slots einer Struktur S verallgemeinerte Variablen sind, können deren Inhalte mit (Setf (<selektorfunktion> s) <wert>) geändert werden.

An dieser Stelle muß unbedingt darauf hingewiesen werden, daß Defstruct ein viel mächtigeres Werkzeug ist, als durch diesen Kurs gezeigt werden kann. So ist es möglich,

```
;;; Definition einer Zeitstruktur aus
;;; Stunde (h) und Minute (m)
(defstruct time (h 0) (m 0))

;;; Definition einer Datumsstruktur aus
;;; Wochentag (wd) Tag (day) und Monat (mon)
(defstruct date (wd 'mo) (day 1) (mon 'jan))

;;; Definition eines Termins aus Beginn
;;; (start), Ende (stop) und Inhalt (content)
(defstruct ap start stop content)

;;; Definition eines Tageseintrags aus
;;; Datum (date) und Terminliste (apl)
(defstruct day date apl)
```

Bild 15 Definition der Strukturen für den Terminplaner

```
(defun next-day
       (date)
 (let ((nwd (get (date-wd date)
                 'next-day))
       (mon (date-mon date))
       (day (date-day date)))
      (cond ; ? letzter Tag des Monats ->
            ;   neuer  Monat
            ((= day (get mon 'day-nr))
             (make-date :wd nwd :day 1
                :mon (get mon 'next-mon)))
            (t (make-date :wd nwd
                  :day (+ day 1)
                  :mon mon)))))
```

Bild 16 Definition der Funktion Next-day

durch Angabe bestimmter Schlüsselwörter den Namen der Konstruktor- und Selektorfunktionen festzulegen, Slots von anderen Strukturen zu erben oder in gewissen Grenzen die physische Repräsentation der Struktur zu bestimmen. Für eine weitergehende Beschreibung sei auf die Sprachdefinition von Common Lisp verwiesen.

In Bild 15 ist nun dargestellt, wie die Strukturen für den Terminplaner definiert werden können.

Als nächstes soll die Funktion Next-day definiert werden, die zu einem gegebenen Datum den Nachfolger berechnet. Dazu müssen die Nachfolger der einzelnen Wochentage, der Monate und die Zahl der Tage im Monat bekannt sein.

Um dieses Wissen abzuspeichern, wird eine weitere Eigenschaft der Symbole ausgenutzt. Neben Werten und einer funktionalen Bedeutung kann man ihnen auch beliebige andere Eigenschaften zuordnen. Um diese Eigenschaften aufnehmen zu können, besitzt jedes Symbol eine sogenannte Eigenschaftsliste property list oder P-Liste, in der die Eigenschaften als Paare von *Indikator* und *Wert* abgespeichert werden können. Als Indikatoren dürfen nur Symbole verwendet werden; als Werte jedoch können beliebige Lisp-Objekte dienen. Der Zugriff zu einer bestimmten Eigenschaft erfolgt über die Funktion Get mit der folgenden Syntax:

(Get <symbol> <indikator>)

Mit (**setf (get** <symbol> <indikator>) <wert>) kann dem Symbol eine Eigenschaft zugeordnet werden.

Unter den Indikator Next-day wollen wir dem Symbol Sonntag das Symbol Montag zuordnen:

(setf (get 'sonntag 'next-day) 'montag)

Für die anderen Wochentage sollen analoge Zuweisungen gelten.

Übung 5: Legen Sie in den P-Listen der Monatsnamen unter den Indikatoren Next-mon und Day-nr die Namen der Folgemonate bzw. die Anzahl der Tage des entsprechenden Monats ab.

```
(defun make-calendar
       (person wd-start day-start
        month-start day-stop month-stop)
 (setf (get person 'calendar)
    (make-calendar-aux day-stop month-stop
       (make-date :wd wd-start
          :day day-start :mon month-start)
          nil)))

(defun make-calendar-aux
       (day-stop month-stop date acc)
 (cond ; Kalender fertig ?-> Umdrehen
       ; des Akkumulatorinhalts
       ((and (eq (date-mon date) month-stop)
           (= (date-day date) day-stop))
        (reverse
           (cons (make-day :date date) acc)))
       ; Restrekursion
       (t (make-calendar-aux day-stop
             month-stop (next-day date)
             (cons (make-day :date date)
                acc)))))
```

Bild 17 Definition der Funktion Make-calendar

```
(defun free-time
       (start stop apl)
 (cond  ;apl leer, dann Termin frei
        ((null apl)
         (list (make-ap :start start
                        :stop stop)))
        ;falsche Parameter
        ((t>= start stop) nil)
        ;Start liegt hinter erstem Termin
        ;aus apl
        ((t>= start (ap-start (car apl)))
         (free-time (if (t>= start
                           (ap-stop
                              (car apl)))
                        start
                        (ap-stop (car apl)))
                    stop (cdr apl)))
        ;Start liegt vor erstem Termin in apl
        ((t>= (ap start (car apl)) stop)
         (list (make-ap :start start
                  :stop stop)))
        ;Rekursion
        (t (cons (make-ap :start :start
                     :stop (ap-start (car apl)))
             (free-time (ap-stop (car apl))
                        stop (cdr apl))))))
```

Bild 18 Definition der Funktion Free-time

Mit den in den P-Listen abgelegten Informationen kann nun die Funktion Next-day definiert werden (Bild 16).

Die Funktion Make-calendar (Bild 17) legt einen leeren Terminplan für einen gegebenen Zeitraum unter dem Indikator Calendar in der P-Liste des Eigentümers ab. Da die meisten Lisp-Systeme auf 16-Bit-Rechnern nicht über Funktionen zur Berechnung des Datums verfügen, muß neben Anfangs- und Endtag und dem Monat auch noch der Wochentag des Anfangsdatums angegeben werden.

Aufgabe 7: Definieren Sie die Funktion Find-day-entry. Die Funktion soll 3 Parameter, den Tag, den Monat und die Kalenderstruktur haben und den entsprechenden Tageseintrag oder NIL liefern.

Zum Finden eines gemeinsamen Termins muß in den Terminplänen aller beteiligten Personen gesucht werden. Zunächst erzeugen wir eine Liste von freien Terminen pro Person, und dann wird der gemeinsame Termin aus dem Durchschnitt der entsprechenden Listen ermittelt. Die Liste der freien Termine wird durch die Funktion Free-Time (Bild 18) gebildet. Die Definition bedarf keiner besonderen Erklärung, nach dem Test von Grenzfällen wird die Liste rekursiv analysiert.

```
(defun common-free
       (apl1 apl2)
 (cond ((null apl1) nil)
       ((null apl2) nil)
       ;Termin in apl1 liegt vor Termin
       ;in apl2
       ((t< (ap-start (car apl2))
            (ap-start (car apl1)))
        (common-free apl2 apl1))
       ;Stop apl1 vor Start apl2 ->
       ;reduziere apl1
       ((t< (ap stop (car apl2))
            (ap-start (car apl1)))
        (common-free (cdr apl1) apl2))
       ;Stop von apl2 vor Stop apl1 ->
       ;(car apl2) ist frei
       ((t< (ap-stop (car apl2))
            (ap-stop (car apl1)))
        (cons (car apl2)
              (common-free apl1
                 (cdr apl2))))
       ;Überlappung
       (t (cons (make-ap :start (ap-start
                                  (car apl2))
                         :stop (ap-stop
                                  (car apl1)))
             (common-free (cdr apl1)
                           apl2)))))
```

Bild 19 Definition der Funktion Common-free

```
(defun common-time
       (p1 p2 day month dur begin end)
 (common-time-aux
     (common-free
        (free-time begin end
             (day-apl
                (find-day-entry day
                   month
                   (get p1 'calendar))))
        (free-time begin end
             (day-apl
                (find-day-entry day
                   month
                   (get p2 'calendar)))))
     dur nil))

(defun common-time-aux
       (apl dur acc)
 (cond ((null apl) (reverse acc))
       ((>= (duration (car apl)) dur)
        (common-time-aux
           (cdr apl) dur (cons
                          (car apl) acc)))
       (t (common-time-aux
             (cdr apl) dur acc))))
```

Bild 20 Definition der Funktion Common-time

Aufgabe 8: Definieren Sie die Prädikate T>= und T<, die die Erfüllung der entsprechenden Ordnungsrelationen für zwei Zeitstrukturen T1 und T2 testen.

Die Funktion Common-free (Bild 19) erzeugt die Liste der gemeinsam freien Termine. Die Definition sichert, daß der erste freie Termin in APL1 vor dem ersten freien Termin in APL2 beginnt. Dann werden drei Standardfälle abgetestet: das erste Element von APL2 liegt hinter dem ersten Element von APL1, das erste Element von APL2 liegt vollständig im ersten Element von APL1, und die Elemente überlappen sich teilweise.

Jetzt fehlt nur noch eine Steuerfunktion zum Aufruf von Common-free, die die Terminpläne zweier Personen bereitstellt und die gemeinsame Liste freier Termine nach Eintragungen durchsucht, die größer als die in der Variablen *DUR* angegebene Dauer in Minuten sind (Bild 20). Die eigentliche Arbeit wird dabei von der restrekursiven Funktion Common-time-aux geleistet.

wird fortgesetzt

Lisp

Teil 3

Rainer Rosenmüller

Dieser Teil des Lehrgangs behandelt die Iteration – für alle, die nach dem streng rekursiven Teil 2 bei der etwas konventionelleren Iteration Abwechslung suchen –, datengesteuerte Programmiertechniken, um die Vorteile von Lisp voll ausschöpfen zu können, die Ein- und Ausgabe sowie Readmakros und Makros zum Schreiben lesbarer Programme. Außerdem gibt es für die Leser(innen), die ihre Termine ohne Computer nicht mehr in den Griff bekommen, die Fortsetzung des Terminkalenderprogramms.

Datengesteuerte Programmierung

Die Iteration

Die *natürliche* Programmiermethode in Lisp ist die Rekursion. Wie in allen Programmiersprachen wird aber auch in Lisp die Iteration durch bestimmte Sprachelemente unterstützt. Die am häufigsten benutzten sind **Dotimes** und **Dolist**. Dotimes ist ein einfacher Zyklus über ganze Zahlen, wie es ihn in allen Sprachen gibt:

(dotimes (<var> <countform> [<resultform>]**)** . <progbody>**)**

Die Form Countform wird ausgewertet und muß eine ganze Zahl als Endwert liefern. Dann läuft die Variable Var von einschließlich Null bis ausschließlich des berechneten Endwerts. Mit jeder dieser Variablenbelegung werden dann die Formen in Progbody abgearbeitet. Zum Schluß wird das Endergebnis durch die Auswertung von Resultform bestimmt. Ist diese nicht angeben worden, so ist das Ergebnis NIL. Ist der Endwert Null oder negativ, so wird der Körper nicht abgearbeitet. Die zu Dotimes analoge Lispform ist Dolist:

(dolist (<var> <listform> [<resultform>]**)** . <progbody>**)**

Dabei wird Listform als erstes ausgewertet und muß eine Liste ergeben. Danach nimmt die Variable Var nacheinander die Werte aller Elemente der Liste an, und der Programmkörper wird mit dem jeweils aktuellen Wert abgearbeitet. Damit kann man die in Bild 22 angegebene – allerdings nicht sehr elegante – Variante der schon bekannten Funktion Reverse schreiben. Diese Konstrukte sind in ihrer Argumentliste sehr einfach und genügen nicht immer den Anforderungen, die sich aus komplexeren Aufgabenstellungen ergeben. Deshalb gibt es noch eine weitere, besser lesbare Form, die auch denen der konventionellen Sprachen sehr nahe kommt:

(do ({(<var> [<initial-value> [<step>]])}**)**
(<endtest> {<result-form>})
{<body-form>}**)**

Die Variablen werden durch die Initialisierungsformen bzw. durch NIL initialisiert und bei jedem Durchlauf durch die Form Step weitergestellt. Ist die Bedingung Endtest erfüllt, werden die Resultformen abgearbeitet und die Schleife wird mit dem Ergebnis der letzten Form verlassen; im anderen Fall werden die Body-Formen abgearbeitet (Bild 23).

```
(defun mp-reverse2 (list)
       (do ((templist list (cdr templist))
            (res nil (cons (car templist) res)))
           ((null templist) res)
           "der koerper ist leer"))
```

Bild 23

Weiterhin gibt es die Form Do*, wobei die Bindung der Variablen nicht mit Let, sondern mit Let* realisiert wird.

Einen zweiten Typ der Iterationskonstrukte stellt Loop dar. Loop ist in seiner Form nicht so festgelegt, wie das oben beschriebene Do. Seine Variabilität erfordert aber auch etwas mehr eigene Programmierung.

(loop {<form>}**)**

Die Formen, die im Körper der Loop-Anweisung stehen, werden wiederholt sequentiell abgearbeitet. Verlassen wird die Loop-Anweisung durch ein Return. Achtung! In Mulisp hat Loop eine etwas exotische Syntax.

Die Funktion Reverse sieht dann wie in Bild 24 dargestellt aus. Bei all diesen Formen handelt es sich jedoch nicht um Funktionen, sondern um Spezialformen bzw. Makros.

```
(defun mp-reverse3 (list)
  (let ((res nil))
       (loop (when (null list) (return res))
             (setq res (cons (car list) res))
             (setq list (cdr list)))))
```

Bild 24

Bevor gezeigt wird, wie Sie sich selbst Makros schreiben können, möchte ich Ihnen noch die explizite Auswertung und einige lispspezifische Iterationsformen erläutern.

Die explizite Auswertung

Die strukturelle Gleichheit von Daten und Programmen macht eine einfache automatische Konstruktion von Programmen möglich und erleichtert das Schreiben von programmerzeugenden Programmen. Die so erzeugten Programme können beispielsweise Werte von Symbolen sein. Die Abarbeitung solcher Programme wird erst durch den expliziten Aufruf der Funktion **Eval** angewiesen. Eval wertet den Ausdruck in der aktuellen dynamischen und in einer leeren lexikalen Umgebung aus. Beinhaltet beispielsweise die Variablen a das *Programm* (+ 1 2 3), so können Sie dieses wie folgt benutzen.

(setq a '(+ 1 2 3)) ==> (+ 1 2 3)
(cons '(das ergebnis ist) a)
==> ((DAS ERGEBNIS IST) + 1 2 3)
(cons '(das ergebnis ist) (eval a))
==> ((DAS ERGEBNIS IST) . 6)

Hierbei müssen sie aber beachten, daß das Argument von Eval zweimal ausgewertet wird. Zuerst wertet der Interpreter (wie für jede Funktion) das Argument aus. Danach verlangt die Funktion Eval, das das übergebene Argument noch einmal ausgewertet wird.

Ein großer Vorteil der Sprache Lisp ist darin zu sehen, daß man mit nur geringem Aufwand erweiterbare Funktionen schreiben kann. Wollen Sie beispielsweise eine Funktion *Typep* schreiben, die prüfen soll, ob ein Lisp-Objekt zu einem bestimmten Datentyp gehört, so müßten Sie diese Funktion ständig ändern, weil Sie sich mit Deftype und Defstruct ständig neue Typen definieren können. Schreiben Sie sich aber die Funktion Typep so, daß sie durch Defstruct oder anderen Funktionen physisch erweitert wird, so könnten Sie Typep nie kompilieren. Diesen Nachteil, den man mit der Erweiterbarkeit einer Funktion in Kauf nehmen muß, kann man durch die datengesteuerte Programmierung umgehen. Wenn Sie in der P-Liste des Symbols Typep unter dem Indikator des Typs die Typprüffunktion ablegt, so könnten die oben genannten Funktionen diese Liste ständig erweitern und das Problem wäre gelöst. Die dafür noch benötigte Funktion heißt Funcall und hat folgende allgemeine Form:

(funcall <fun-object> <args>**)**

Diese Funktion ruft die Funktion Fun-objekt auf und übergibt ihr die restlichen Argumente als aktuelle Parameter. Die P-Liste von Typep könnte etwa den folgenden Inhalt haben:

(string #'stringp stream #'streamp
ship #'(lambda (objekt)
(and (structurep object)
(eq (car object) 'ship))) ...)

Und so könnte Typep folgendermaßen definiert werden:

(defun mp-typep (object type)
(funcall (get 'typep type) object))

Damit ist die Funktion Typep natürlich noch nicht vollständig definiert, aber ein wesentlicher Teil ist schon geschafft. Das in der P-Liste auftretende #'streamp ist eine Abkürzung für (function streamp). Nur liefert *Function* (#') im Unterschied zu Quote (') die funktionale Bedeutung des Symobols, einschließlich der für die Funktion notwendigen Lisp-Umgebung, und nicht das Symbol selbst als Ergebnis. Werden die Argumente einer Funktion nicht einzeln, sondern bereits in einer Liste übergeben, müssen Sie allerdings anstelle von Funcall die Funktion Apply benutzen. Die allgemeine Form lautet:

(apply <fun-object> <arg> <more-args>**)**

Diese Funktion arbeitet ähnlich wie Funcall; das letzte Argument kann aber auch eine Liste sein, die an die Liste der davor angegebenen Argumente angehängt wird.

(apply #'+ 3 4 '(6 8)) ==> 21

Die Lambda-Funktion und die Lambda-Liste

In der P-Liste von Typep ist aber noch eine weitere, unbenannte Funktion enthalten. Die allgemeine Form dafür ist:

```
(defun palindromp (string)
   (let ((start 0) (end (length string)))
        (dotimes (i (floor (- end start) 2) t)
                 (unless (char-equal (char string (+ start i))
                                     (char string (- end i 1)))
                         (return nil)))))

(palindromp "Marktkram") ==> T
(palindromp "Das ist kein Palindrom") ==> NIL
```

Bild 21

```
(defun mp-reverse1 (list)
   (let ((newlist nil))
        (dolist (element list newlist)
                (setq newlist (cons element newlist))))))
```

Bild 22

(lambda <lambda-list> {<declaration>} {<form>}**)**
Die allgemeine Form der Lambda-Liste zeigt Bild 25.

```
({<required-var>}
 [&optional {<optional-var> |
              (<optional-var> <initial-value>)}]
 [&rest <rest-var>]
 [&key
       {<key-var> |
       ({<key-var> |
       (<keyword> <key-var>)} <initial-value>)}])
```

Bild 25

Wenn solch eine Form auf Argumente angewendet wird, so werden erst die Variablen der Lambda-Liste lexikalisch gebunden, und anschließend werden die Formen sequentiell abgearbeitet. Das Ergebnis ist das der letzten Form. Die Bindung der Variablen des Variablenteils geschieht in folgender Reihenfolge: Zuerst werden die Variablen der obligaten Parameter gebunden. Danach werden die Variablen der optionalen Parameter an die angegebenen Werte (die Werte der Initialisierung bzw. NIL, wenn keine Anfangswerte gegeben wurden) gebunden. Anschließend wird die Restvariable an die Liste der noch verbleibenden Argumente gebunden. Die Schlüsselwortvariablen werden zum Schluß gebunden. Diese sind keine Stellungsparameter, und die Zuordnung Variable-Wert erfolgt durch die paarweise Angabe Schlüsselwort-Wert. Unbenannte Funktionen können an allen Stellen stehen, an denen Funktionen verlangt werden.
((lambda (a b) (* a b)) 4 5) ==> 20

Die Map-Funktionen

Diese Funktionen sind die lisptypischen Iterationskonstrukte. Zu der Gruppe gehören Mapc, Mapcan, Mapcar, Mapl, Maplist und Mapcon; ihre allgemeine Form sieht folgendermaßen aus:
(mapxxx <function> <list> <mor-lists>**)**
Allen Funktionen ist gemeinsam, daß sie die übergebene Funktion (1. Argument) auf ihre Argumentliste(n) anwenden und daß sie die Argumentliste(n) nach jeder Anwendung auf den Cdr der Argumentliste(n) reduzieren. Es müssen so viele Argumentlisten angegeben werden, wie die Funktion (1. Argument) als Argumente benötigt. Die Wirkungsweise der Map-Funktionen läßt sich nach der Art der Argumentübergabe an die Funktion und der Ergebnisbildung unterscheiden.
Die Funktionen Mapc, Mapcar und Mapcan übergeben an ihre Argument-Funktion jeweils den Car der aktuellen Argumentlisten, während die Funktionen Mapl, Maplist und Mapcon jeweils die gesamte aktuelle Argumentliste übergeben.
Die Funktionen Mapc und Mapl ignorieren dabei die Ergebnisse der Iterationsschritte. Die Funktionen Mapcar und Maplist bilden eine Liste der Ergebnisse aller Iterationsschritte mittels Cons und die Funktionen Mapcan und Mapcon bilden eine Liste aller positiven Ergebnisse (ungleich NIL) der Iterationsschritte mittels Nconc. Die Ergebnisse von Mapc und Mapl hängen vom Lisp-Dialekt ab. In den meisten alten Dialekten ist der Wert NIL aber in Common Lisp ist er gleich dem zweiten Argument. Daraus ergeben sich die typischen Anwendungen der Funktionen: Mapc und Mapl werden zur Erzielung von Seiteneffekten benutzt, Mapcan und Mapcon werden als Filterfunktionen benutzt.
(Anmerkung: Die Funktion Mapl kann auch manchmal Map heißen!) Zum Beispiel kann man eine Liste der Elemente einer Ausgangsliste bilden, die einen Test erfüllen, indem man folgendes schreibt:
(mapcan #'(lambda (x)
(if (test x) (list x) nil)) liste1)
Man kann auch leicht zwei Listen zu Paaren mischen:
(mapcar #'cons '(1 2 3)
'(auswahl-1 auswahl-2 auswahl-3))
==> ((1 . AUSWAHL-1) (2 . AUSWAHL-2) (3 . AUSWAHL-3))

Einfache Eingaben

Die wichtigsten Eingabefunktionen sind Read-char, Read-line und Read. Für alle Funktionen in Common Lisp gilt die folgende Parameterliste:
&optional (<stream> nil**)**
(<eof-error-p> **t)** <eof-value>
(<recursive-p> nil)
Die hier angeführten Streams sind Lisp-Objekte, die die Verbindung zwischen Lisp und dem Betriebssystem des Rechners herstellen. Die Standardstreams sind beim Start des Systems bereits definiert, können aber vom Lispnutzer neu gebunden werden. Das Argument Stream kann somit NIL für den Wert der Variablen *STANDARD-INPUT* als Eingabe-Stream, T für *TERMINAL-IO* oder ein beliebiger Eingabe-Stream sein. Das Argument Eof-error-p zeigt an, ob eine Fehlerbehandlung ausgelöst werden soll, wenn während des Lesens das Dateiende (EOF) erreicht wurde. Ist Eof-erro-p gleich NIL, so wird bei Erreichen des Dateiendes Eof-value als Ergebnis des Lesevorgangs zurückgegeben. Das Ergebnis der Funktion Read-char ist ein einzelnes Zeichen (als Lisp-Objekt), das von Read-line ist eine Zeichenkette, die die Zeichen der nächsten Zeile des Streams (ohne New-line) enthält, und zusätzlich als zweiten Wert T oder NIL als Zeichen dafür, ob das abschließende Zeichen der Zeile ein EOF war oder nicht. Die Funktion Read dagegen liest den nächsten Lisp-Ausdruck (in der Lisp-Syntax unabhängig von der physischen oder logischen Strukturierung des Streams).
(read-char)x
==> #\x
(read-line)Das ist eine Zeile
==> "Das ist eine Zeile"
(read) (a b c #\Space 1 2.4 "otto")
==> (A B C #\Space 1 2.4 "otto")
Die Funktionen Read-char und Read-line können auch dazu benutzt werden, lispunabhängige, nutzereigene Eingaben einzulesen.

Einfache Ausgaben

Die wichtigsten Ausgabefunktionen lauten in ihrer allgemeinen Form:
(prin1 <object> **&optional** <stream>**)**
(princ <object> **&optional** <stream>**)**
(terpri &optional <stream>**)**
Das Argument Stream hat die gleiche Funktion wie bei der Eingabe. Die Funktion Terpi (terminate print-line) gibt ein Newline aus. Die Funktion Prin1 gibt die externe Darstellung eines Lisp-Objektes in der vollständigen Lisp-Syntax aus, das heißt, diese Form kann durch Read wieder eingelesen werden, dagegen gibt Princ eine verkürzte, lesbare Darstellung aus.
(prin1 '(a #\b "abc\\xyz\"123")
==> (A #\b "abc\\xyz\"123")
(princ '(a #\b "abc\\xyz\"123")
==> (A b abc\xyz"123)
Als Ergänzung gibt es noch die Funktion Print. Sie gibt ein New-line aus, ruft dann Prin1 auf und gibt danach ein Leerzeichen aus. Manche Lisp-Dialekte haben noch die Funktionen Pprint für eine strukturierte Ausgabe der Lisp-Objekte und Format für die formatierte Ausgabe.
Alle Printfunktionen werden in ihrer Wirkung durch globale Steuervariablen weiter spezifiziert. So bestimmt zum Beispiel die Variable *PRINT-BASE*, zu welcher Basis die rationalen Zahlen ausgegeben werden. Die Variablen *PRINT-LEVEL* und *PRINT-LENGTH* begrenzen die Tiefe und die Länge von auszugebenden Listen, wenn sie nicht den Wert NIL haben.
Ist *PRINT-LEVEL* eine positive ganze Zahl, so werden Listen nur bis zu dieser Tiefe ausgegeben. Alle tiefer gehenden Komponenten der Liste werden durch das Zeichen # angedeutet. Wird die Listenausgabe in der Länge begrenzt, so werden nur so viele Elemente ausgegeben, wie verlangt werden. Ist die Liste länger, so wird der Rest durch die Zeichen ... abgekürzt. Das kann sehr nützlich sein, wenn man große oder zyklische Listen ausgeben will, um sie zu überwachen oder zu identifizieren.
Das folgende Beispiel soll das verdeutlichen.
(setq *print-level* 2) ==> 2
(setq *print-length* 3) ==> 3
'(a (b c (d e) f) (g h i k) l m)
==> (A (B C # ...) (G H I ...) ...)

Die formatierte Ausgabe

Die allgemeine Form der formatierten Ausgabe lautet:
(format <stream> <control-string-> <args>**)**
Das Argument Stream hat hier eine etwas andere Bedeutung: T steht für *STANDARD-OUTPUT* und NIL bewirkt, daß von Format eine Zeichenkette erzeugt wird, die die ausgegebenen Zeichen enthält. Die Ausgabe entsteht folgendermaßen. Die Zeichen, die in control-string stehen, werden nacheinander ausgegeben. Wird das Zeichen Tilde ~ erkannt, so beginnt eine Formatanweisung. Diese Anweisung wird analysiert und ausgeführt, wobei Elemente aus der restlichen Argumentliste Args verbraucht werden können. Anschließend wird die Steuerzeichenkette weiter abgearbeitet. Die wichtigsten Formatanweisungen sind:

~%	Terpi wird ausgeführt
~s	Prin1 wird mit dem nächsten Argument aufgerufen
~a	Princ wird benutzt

(let ((x '(a b "xyz")))
(format t
"~%Das ist die Liste ~a mit~
~a Elementen~%third ist aber ~s!"
x (length x) (third x)))
==>
Das ist die Liste (A B xyz) mit 3 Elementen
third ist aber "xyz"!

Natürlich ist das nur eine sehr kurze Einführung in die Format-Welt, denn der Umfang

von Format übersteigt die Formatierungsunterstützungen aller anderen bekannten Sprachen.

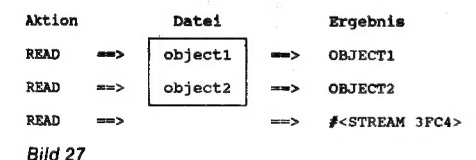
Das Öffnen und Schließen von Dateien

Mit den bisher vorgestellten Funktionen ist nur eine Ausgabe auf den Bildschirm möglich, weil die Funktion Open, zur Erzeugung eines Streams noch fehlt. Diese hat für die Eingabe die allgemeine Form:
(open <filename> :direction :input)
Damit wird einen Stream erzeugt, der Eingaben (:input), Ausgaben (:output) oder Ein- und Ausgaben (:io) ermöglicht und der mit der angegebenen Datei verbunden ist. Der Name muß eine Zeichenkette mit für Dateinamen gültigen Zeichen sein, kann aber in Abhängigkeit vom Betriebssystem und des Lisp-Dialekts stark reduziert sein. Abgeschlossen wird ein Stream durch
(close <stream>)
Die Funktion Load zum Einlesen von Quelltexten könnte dann wie im Bild 26 aussehen.

```
(defun mp-load (filename &key (print nil))
  (format ";LOAD ~a start" filename)
  (do* ((str (open filename :direction :input))
        (obj (read str nil str)
             (read str nil str)))
       ((eq str obj)
        (format ";LOAD ~a end" filename)
        (return t))
       (if print
           (print (eval obj))
           (eval obj))))
```

Bild 26

Am Anfang der Do*-Schleife werden Str und Obj initialisiert, dann wird nur noch Obj in jedem Zyklus neu bestimmt. Interessant ist die Abbruchbedingung. Das Ende der Datei (EOF) wird vom Read erkannt und bewirkt, daß Read als Eof-value den Stream Str, der garantiert nicht eingelesen werden konnte, als Ergebnis liefert. Streams sind Datenobjekte, die keine externe Repräsentation besitzen und somit auch nicht in eine Datei ausgelagert werden können. Damit sind sie aber als eindeutige Markierung für das Ende einer Datei verwendbar und die Iteration kann bei diesem Ergebnis abgebrochen werden (Bild 27).



Aktion		Datei		Ergebnis
READ	==>	object1	==>	OBJECT1
READ	==>	object2	==>	OBJECT2
READ	==>		==>	#<STREAM 3FC4>

Bild 27

Makros und Readmakros

Makros ersetzen normalerweise ein Stück Quelltext durch einen anderen. In dem Read-eval-print-Zyklus gibt es in Lisp zwei Stellen, an denen solche Makros aktiv werden können, beim Read und beim Eval.
Readmakros sind einzelne Zeichen, die mit einer Lesefunktion verknüpft sind. Treten diese im Zeichenstrom auf, so setzt die zugehörige Funktion das Lesen fort und bestimmt auch das Ergebnis der Leseaktion. Die wichtigsten Readmakros sind das Quote-(') und das Backquote-Makro (`). Aus 'form wird beim Einlesen (QUOTE FORM) und aus `form wird (BACKQUOTE FORM). Die Wirkung von Quote ist Ihnen ja bereits bekannt. Die Arbeitsweise des Backquote-Makros ist etwas komplizierter. Normalerweise wertet der Interpreter alle Formen aus; nur mit Quote kann eine Auswertung explizit unterdrückt werden. Innerhalb des Backquotes arbeitet das Lisp gewissermaßen im inversen Interpretermodus: Alle nicht extra markierten Ausdrücke werden als gequotet angesehen, soll eine Auswertung bestimmter Ausdrücke erfolgen, so muß dies explizit angegeben werden. Wie das Ergebnis der Auswertung in den Gesamtausdruck eingefügt werden soll, wird durch die Art der Markierung des Ausdrucks angegeben: Steht ein Komma davor, so wird das Ergebnis so wie es kommt eingebaut, steht ein ,@ davor, so werden die Elemente der bei der Auswertung entstehenden Liste in den Gesamtausdruck einzeln eingebaut. Damit eignen sich Backquote-Ausdrücke sehr gut zum Schreiben von Makros oder zur Erzeugung von Programmen.
(setq x '(* 2 3)) ==> **(* 2 3)**
`(list x y ,x z ,@x w)
==> **(list x y (* 2 3) z * 2 3 w)**
Weitere Readmakros gibt es für Listen, für Zeichenketten, für Kommentare bis zum Zeilenende (;), für die schon bekannte Funktionsabkürzung (#'), für einfache Vektoren (#), für balancierte, schachtelbare Kommentare beliebiger Länge (#| ... |#), für Zeichen (#\), für komplexe Zahlen (#C) und für binäre, oktale und hexadezimale rationale Zahlen (#B, #O und #X), die aber alle sehr dialektabhängig sind. Die meisten Readmakrozeichen sind terminierende Zeichen, das heißt, sie beenden ein angefangenes Lexem. So wird abc'def zu ABC und (QUOTE DEF) eingelesen. Unter den Standardmakrozeichen ist nur das Doppelkreuz (#) nicht terminierend und kann in ein Lexem eingebettet werden.
Die Auswertung der Makros erfolgt (im Gegensatz zu den Readmakros) beim Eval. Wenn der Interpreter eine Form auswertet, deren erstes Element in der Liste eine funktionale Bedeutung besitzt, so werden alle weiteren Objekte ausgewertet und anschließend der Funktion als Argumente übergeben. Liefert die Funktionsauswertung des ersten Elementes aber keine Funktion, sondern eine Spezialform oder ein Makro, so werden die weiteren Objekte unausgewertet an das Makro übergeben, das diese dann nach eigenem Ermessen auswertet oder nicht. In dieser ersten Auswertungsrunde liefert der Makro-Aufruf die sogenannte Makroexpansion. In der zweiten Auswertungsrunde wird nun der vom Makro erzeugte Kode ausgewertet und liefert das endgültige Ergebnis. Im Gegensatz zu anderen Sprachen (C, Assembler) werden in Lisp die Makros also zur Laufzeit ausgewertet. Die allgemeine Form zum Definieren eines Makros ist:
(defmacro <name> <lambda-list>
{<declaration>} {<form>}**)**
Man kann nun leicht ein Makro Numeric-if definieren, das einen numerischen Wert auf gleich, größer oder kleiner Null abtestet und in Abhängigkeit vom Ausgang des Tests entweder die Zf-, Gf- oder die Lf-Form auswertet (Bild 28).

```
(defmacro numeric-if (nv zf gf lf)
  (let ((sym (gensym)))
    `(let ((,sym ,nv))
       (cond ((zerop ,sym) ,zf)
             ((> ,sym 0) ,gf)
             ((< ,sym 0) ,lf)))))
```

Bild 28

Dieses Beispiel demonstriert einen wichtigen Sachverhalt. Wäre der Wert der Auswertung des Arguments nv nicht gebunden, so würde nv in der erzeugten Form dreimal enthalten sein und je nach seinem Wert ein- bis dreimal ausgewertet werden, was – abgesehen vom Zeitverhalten – insbesondere bei Funktionen mit Seiteneffekten zu Fehlern führen kann. Deshalb wird dieser Wert an ein Symbol gebunden, das vorher von der Funktion Gensym bereitgestellt wurde. Die Funktion Gensym erzeugt ein im Lisp-System noch nicht vorhandenes Symbol, das jedoch nicht in die Namenstabelle des Systems eingetragen wird.
Da in Common Lisp If die Basisform ist, kann man Cond durch If beschreiben (Bild 29).

```
(defmacro mp-cond (&rest clauses)
 (let ((test (caar clauses))
       (is-body (cdar clauses))
       (body (if (cddar clauses)
                 `(progn ,@(cdar clauses))
                 (cadar clauses))))
   (if (null clauses)
       nil
       (if is-body
           (if (eq test t)
               body
               `(if ,test ,body
                    (mp-cond ,@(cdr clauses))))
           (let ((sym (gensym)))
             `(let ((,sym ,test))
                (if ,sym ,sym
                    (mp-cond ,@(cdr clauses)))))))))
```

Bild 29

Hierbei ist zu beachten, daß der Körper einer Klausel mehrere Formen enthalten kann, die dann in ein Progn eingekleidet werden müssen, daß eine Klausel keinen Körper haben muß, so daß der Test im positiven Fall auch das Ergebnis liefert, und daß die letzte Klausel den Test T haben kann, so daß keine If-Form mehr erzeugt werden muß.

Das verallgemeinerte Printmakro

Noch einmal zurück zur Ausgabe. In der Literatur wird manchmal ein Printmakro vorgeschlagen, das ähnlich wie Format eine formatierte Ausgabe ermöglicht und vom Nutzer erweitert werden kann. So soll beispielsweise die Anweisung
(let ((a10) (b 20))
(msg nil t
"Der Flächeninhalt des Feldes"
a " x " b
"ist gleich" 5 (* a b) "." t))
folgende Ausgabe erzeugen:
Der Flächeninhalt des Feldes 10 × 20
ist gleich 200.
Das angegebene T wird in einen Terpi-Aufruf umgewandelt, positive Zahlen bewirken die Ausgabe von Leerzeichen, Zeichenketten werden unverändert und andere Lispausdrücke nach erfolgter Auswertung ausgegeben (Bild 30).
Hier werden sowohl Zahlen als auch Schlüsselwörter im Car von Listen zur Steuerung der Formatierung benutzt. Man kann dieses Makro erweiterbar machen, indem man zwei Funktionen Set-msg-method und Get-msg-method definiert. Set-msg-method legt in der P-Liste des Symbols MSG unter einem Indikator, dem Schlüsselwort, die zugehörige Funktion ab. Get-msg-method stellt diese Funktion wieder zur Verfügung. Man könnte diese Informationspaare auch in A-Listen (diese werden im nächsten Teil behandelt) oder in Hashtabellen (die leider nicht in allen

```
(defmacro msg (stream &rest args)
 (let ((sym (gensym)))
  `(let ((,sym ,stream))
    ,@(do* ((argl args (cdr argl))
            (arg (car argl) (car argl))
            (res nil))
           ((null argl) (nreverse res))
        (setq res
         (cons
          (cond ((eq arg 't) `(terpri ,sym))
                ((stringp arg) `(princ ,arg ,sym))
                ((numberp arg)
                 (numeric-if arg
                    `(fresh-line ,sym)
                    `(write-spaces ,sym ,arg)
                    `(write-newlines ,sym
                                     ,(- arg))))
                ((atom arg) `(prin1 ,arg ,sym))
                (t (case (car arg)
                    (:eval (cadr arg))
                    (:lines
                     `(write-newlines ,sym
                                      ,(cadr arg)))
                    (:spaces
                     `(write-spaces ,sym
                                    ,(cadr arg)))
                    (:pp
                     `(pprint ,(cadr arg) ,sym))
                    (otherwise
                     `(prin1 ,arg ,sym)))))
               res))))))

(defun write-spaces (stream count)
     (dotimes (k count) (write-char #\Space)))

(defun write-newlines (stream count)
     (dotimes (k count) (terpri stream)))
```

Bild 30

Dialekten enthalten sind) ablegen. Die Case-Anweisung im T-Zweig des Cond müßte dann wie folgt verändert werden:

```
(t (let ((fn (get-msg-method (car arg))))
     (if fn (apply fn ,sym (cdr arg))
         `(prin1 ,arg ,sym))))
```

Der Terminkalender

Im folgenden möchte ich Ihnen einige mit Hilfe dieses Makros geschriebene Ausgabefunktionen für das Programm des Terminkalenders vorstellen.

Die erste Funktion schreibt eine Terminliste auf den Stream ∗STANDARD-OUTPUT∗, das heißt auf das Terminal (wenn ∗STANDARD-OUTPUT∗ nicht anders gebunden wurde). Die benutzte Funktion, die einen Termin ausgibt, wird wieder auf Funktionen zurückgeführt, die die Zeit bzw. den Inhalt des Termines ausgeben (Bild 31). An diesem Beispiel sieht man einen wesentlichen Vorteil des MSG-Makros gegenüber Format: Die Ausgabefolge kann eigene Printfunktionen enthalten, im Gegensatz zu Format, das eine Funktion (und kein Makro) ist und somit keine verzögerte Auswertung ermöglicht.

Aufgabe 9:

Schreiben Sie für das MSG-Makro eine Methode :when mit zwei Argumenten. Das erste Argument ist ein Test, der ausgewertet wird, das zweite Argument soll aber nur ausgewertet werden, wenn der Test wahr ergeben hat.

Schreiben Sie anschließend damit die Funktion Write-time, die die Uhrzeit im Format "hh.mm Uhr" (wenn notwendig mit führender Null) ausgibt und als Ergebnis liefert.

Die nächste Funktion wird nur der besseren Lesbarkeit des Programmes wegen definiert.

```
(defun write-content (content) (princ content))
```

Zum Eintragen bestehender Termine in den Kalender gibt es die Funktion Update-calendar. Der Planung eines gemeinsamen Termins zweier Personen dient die Funktion Plan-ap (Bild 32).

```
(defun write-apl (apl)
     (mapc #'write-ap apl) apl)

(defun write-ap (ap)
     (msg nil t "von " (:eval (write-time (ap-start ap)))
                "bis " (:eval (write-time (ap-stop ap)))
                1 (:eval (write-content (ap-content ap))))
    ap)
```

Bild 31

```
(defun update-calendar (person)
  (loop (terpri)
        (princ "Datum [dd mmm]: ")
        (new-ap person (read) (read) (read-ap))
        (princ "noch mehr Eintäge ? [j/n] ")
        (if (not (eq (read) 'j)) (return nil))))

(defun read-date ()
     (princ "Datum [dd mmm]: ")
     (make-date :day (read) :mon (read)))

(defun read-ap ()
     (princ "von [hh mm]: ")
     (let ((start (read-time)))
          (princ "bis [hh mm]: ")
          (make-ap :start start :stop (read-time)
                   :content (read-content))))

(defun read-time ()
     (make-time :h (read) :m (read)))

(defun read-content ()
     (princ "Ereignis [mit RETURN abschließen]: ")
     (read-line))

(defun plan-ap ()
   (common-ap
      (progn (terpri) (princ "Person1: ") (read))
      (progn (terpri) (princ "Person2: ") (read))
      (read-date)
      (progn (terpri)
             (princ "Zeitdauer (min): ")
             (read))
      (read-content)))

(defun common-ap (p1 p2 date dur content)
 (let* ((day (date-day date))
        (mon (date-mon date))
        (time (common-time p1 p1 day mon dur
                   *arbeitsbeginn* *arbeitsende*)))
      (if time
          (let* ((first (car time))
                 (start (ap-start first))
                 (ap (if (= (duration first) dur)
                         first
                         (make-ap :start start
                            :stop
                            (compute-end start
                                         dur)
                            :content content))))
                (new-ap p1 day mon ap)
                (new-ap p2 day mon ap))
          nil)))
```

Bild 32

Aufgabe 10:

Schreiben Sie die (iterative) Funktion Compute-end, die die Zeitstruktur Time und die Zeitdauer in Minuten bekommt und die um diesen Zeitraum verschobene Zeit berechnet. Die Funktion New-ap mit den Argumenten Person, Tag, Monat und Termin, die den neuen Termin in den Kalender der Person eintragen soll, können Sie mit der Funktion Find-day-entry und der Funktion Insert-ap schreiben. Die Funktion Insert-ap soll die Argumente Termin und Terminliste bekommen und eine neue Terminliste mit dem richtig einsortierten Termin erzeugen. Mit der Funktion Save-calendar kann man die Kalender einer Liste von Personen auf ein File schreiben, das man später mit Load wieder einlesen kann, um mit den gesicherten Daten weiterarbeiten zu können (Bild 33).

wird fortgesetzt

```
(defun save-calendar (filename &rest persons)
 (let ((*standard-output*
         (open filename :direction :output)))
      (mapc #'save-calendar1 persons)
      (close *standard-output*)))

(defun save-calendar1 (person)
     (terpri)
     (pprint `(setf (get ',person 'calendar)
                    ',(get person 'calendar)))
     (terpri))
```

Bild 33

```
Aufgabe 9:
(:when `(if ,(cadr arg) ,(caddr arg)))

Die Funktion Write-time kann so aussehen:
(defun write-time (time)
 (msg nil (:when (< (time-h time) 10) (princ 0))
          (:eval (princ (time-h time)))
          (:when (< (time-m time) 10) (princ 0))
          (:eval (princ (time-m time)))  " Uhr")
 time)

Aufgabe 10:
(defun compute-end (time dur)
 (do* ((h (time-h time) (1+ h))
    (m (time-m time) 0)
    (du dur (- du rest))
    (rest (- 60 m) (- 60 m)))
   ((< du rest) (make-time :h h :m (+ m du)))))

(defun new-ap (person day mon ap)
 (let ((entry
         (find-day-entry day mon
         (get person 'calendar))))
       (setf (day-apl entry)
         (insert-ap ap (day-apl entry)))
       entry))

(defun insert-ap (ap apl)
 (cond ((null apl) (list ap))
       ((t< (ap-start (car apl))
            (ap-start ap))
        (cons (car apl) (insert-ap ap (cdr apl))))
       (t (cons ap apl))))
```

Bild 34 Lösungen der Aufgaben 9 und 10

```
Aufgabe 4:
(defun mp-length (li)
 (cond ;;Abbruchtest
       ((null li) 0)
       ;;Rekursion
       (t (+ 1 (mp-length (cdr li))))))

Aufgabe 5:
(defun mp-reverse(li)
  (mp-reverse-aux li nil))

(defun mp-reverse-aux (li acc)
  (cond ;;Abbruchtest
        ((null li) acc)
        ;;Restrekursion
        (t (mp-reverse-aux (cdr li)
                  (cons (car li) acc)))))

Aufgabe 6:
(defun mp-reverse(li)
 (declare (special li))
 (let (res)
       (declare (special res))
       (mp-reverse-aux li)
      res))

(defun mp-reverse-aux(l)
 (when l ;;Rückwärtsrekursion
         (mp-reverse-aux (cdr l))
         (setq res (cons (car li) res)
               li (cdr li))))

Aufgabe 7:
(defun find-day-entry (day month cal)
 (cond ;;Abbruch
       ((null cal) nil)
       ;;Definition lokaler Variable
       ;;für Rekursion
       (t (let* ((entry (car cal))
                 (date (day-date entry)))
               (cond ;;Datum gefunden ?
                     ((and
                        (eq (date-mon date)
                             month)
                        (= (date-day date)
                            day))
                      entry)
                     ;;Rekursion
                     (t (find-day-entry day
                          month
                           (cdr cal))))))))

Aufgabe 8:
(defun t>= (t1 t2)
 (or (> (time-h t1) (time-h t2))
     (and (= (time-h t1) (time-h t2))
          (>= (time-m t1) (time-m t2)))))

(defun t< (t1 t2)
 (or (< (time-h t1) (time-h t2))
     (and (= (time-h t1) (time-h t2))
          (< (time-m t1) (time-m t2)))))

Die Funktion Duration:
(defun duration (ap)
 (+ (* 60 (- (time-h (ap-stop ap))
             (time-h (ap-start ap))))
    (- (time-m (ap-stop ap))
       (time-m (ap-start ap)))))
```

Bild 35 Lösungen der Aufgaben 4 bis 8 aus dem Teil 2